BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EFFICIENT COMMUNICATION IN MULTI-GPU SYSTEMS

EFEKTIVNÍ KOMUNIKACE V MULTI-GPU SYSTÉMECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. MATEJ ŠPEŤKO**
AUTOR PRÁCE

SUPERVISOR — **Ing. FILIP VAVERKA**
VEDOUCÍ PRÁCE

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2017/2018

Zadání diplomové práce

Student: **Špeťko Matej, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **Efektivní komunikace v multi-GPU systémech**
Efficient Communication in Multi-GPU Systems
Kategorie: Paralelní a distribuované výpočty
Zadání:

1. Seznamte se s možnými způsoby přenosu dat mezi GPU akcelerátory. Vezměte v úvahu jak systémy se sdíleným adresovým prostorem, tak distribuované systémy (clustery).
2. Osvojte si prostředí a knihovny umožňující řízení komunikace mezi akcelerátory a uzly distribuovaných systémů.
3. Analyzujte metody komunikace dostupné na superpočítači Anselm a porovnejte jejich vlastnosti.
4. Zvolte vhodnou vědeckou aplikaci, na které bude možné ukázat vliv zvolených metod komunikace.
5. Implementujte zvolenou aplikaci za použití vybraných metod komunikace.
6. Experimentálně vyhodnoťte vlastnosti implementované aplikace. Primárně se zaměřte na datové přenosy.
7. Zhodnoťte dosažené výsledky, diskutujte vlastnosti implementovaných metod komunikace a možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a popis jednotlivých etap řešení. Odevzdává se v elektronické podobě a ve dvou výtiscích, přičemž oba musí obsahovat podepsané prohlášení o autorství. Jeden výtisk musí být svázán nerozebíratelným způsobem.

Vedoucí práce: **Vaverka Filip, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 11. července 2018
Datum odevzdání: 31. července 2018

Abstract

After the introduction of CUDA by Nvidia, the GPUs became devices capable of accelerating any general purpose computation. GPUs are designed as parallel processors which posses huge computation power. Modern supercomputers are often equipped with GPU accelerators. Sometimes single GPU performance is not enough for a scientific application and it needs to scale over multiple GPUs. During the computation, there is a need for the GPUs to exchange partial results. This communication represents computation overhead and it is important to research methods of the effective communication between GPUs. This means less CPU involvement, lower latency and shared system buffers. This thesis is focused on inter-node and intra-node GPU-to-GPU communication using GPUDirect technologies from Nvidia and CUDA-Aware MPI. Subsequently, k-Wave toolbox for simulating the propagation of acoustic waves is introduced. This application is accelerated by using CUDA-Aware MPI. Peer-to-peer transfer support is also integrated to k-Wave using CUDA Inter-process Communication.

Abstrakt

Po predstavení CUDA technológie od Nvidie môžu byť na grafických kartách počítané všeobecné výpočty. Grafické karty sú v podstate paralelné procesory s vysokým výpočtovým výkonom. Moderné superpočítače bývajú vybavené grafickými kartami ako akcelerátormi. Pri niektorých aplikáciách však výkon jednej grafickej karty nestačí a ich výpočet musí byť rozdelený medzi niekoľko grafických kariet. Počas výpočtu je potrebné vymieňať medzi grafickými kartami čiastkové výsledky. Táto komunikácia značne brzdí výpočet a preto je potrebné skúmať metódy efektívnej komunikácie medzi grafickými kartami – metódy ktoré menej zapájajú CPU, znižujú odozvu a zdieľajú systémové zásobníky. V tejto diplomovej práci je skúmaná komunikácia grafických kariet v rámci jedného uzla aj v rámci celého superpočítača. Hlavný dôraz je na technológie GPUDirect od Nvidie a CUDA-Aware MPI. Následne je predstavený k-Wave toolbox, aplikácia pre simuláciu šírenia akustických vĺn. Táto aplikácia je akcelerovaná pomocou CUDA-Aware MPI. Do tejto aplikácie je taktiež pridaná podpora peer-to-peer prenosov pomocou CUDA Inter-process Communication.

Keywords

CUDA, MPI, GPUDirect, RDMA, CUDA-Aware MPI, CUDA IPC, Anselm, HPC, GPGPU, peer-to-peer, k-Wave.

Kľúčové slová

CUDA, MPI, GPUDirect, RDMA, CUDA-Aware MPI, CUDA IPC, Anselm, HPC, GPGPU, peer-to-peer, k-Wave.

Reference

ŠPEŤKO, Matej. *Efficient Communication in Multi-GPU Systems*. Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Filip Vaverka

Rozšírený abstrakt

Po predstavení CUDA technológie od Nvidie sa grafické karty (GPU) zmenili zo zariadení, ktoré vykresľujú obraz na zariadenia schopné akcelerovať ľubovoľný výpočet. GPU sú v princípe masívne paralelné procesory. Z tohto dôvodu dosahujú aplikácie akcelerované GPU vyššieho výpočtového výkonu ako keby bežali len na CPU. Navyše, GPU majú aj vyššiu účinnosť a dosahujú nižšiu spotrebu energie ako CPU s porovnateľným výkonom. Vedecká komunita si grafické akcelerátory obľúbila pre ich výhody a dnes je bežné sa s nimi stretnúť pri vysoko náročných výpočtoch.

Existuje však aj veľa vedeckých aplikácii, ktorým výkon jedného GPU nestačí. Alebo je objem dát spracovávaných aplikáciou tak veľký, že nevojde do pamäte grafickej karty. Z tohto dôvodu sa budujú aplikácie, ktoré vedia bežať na viacerých GPU. Niektoré aplikácie ako napríklad lámanie hesiel nevyžadujú veľký objem komunikácie počas výpočtu. No na druhej strane sú aplikácie, ktorých výkon je limitovaný práve výmenou dát medzi GPU. Z tohto dôvodu je nevyhnutné skúmať efektívne metódy dátových prenosov medzi GPU.

V mojej diplomovej práci skúmam metódy efektívnej komunikácie z GPU do GPU. To znamená metódy, ktoré pri prenose nezaťažujú CPU, znižujú prístupovú dobu, majú priamy prístup do GPU pamäte a využívajú spoločné pamäťové zásobníky pre grafickú a sieťovú kartu. Metódy zahŕňajú komunikáciu v distribuovanom prostredí ako aj v rámci jedného servera s viacerými GPU. Hlavný dôraz je kladený na GPUDirect technológie od Nvidie a CUDA-Aware MPI.

V mojej práci je ďalej predstavená open source aplikácia k-Wave toolbox, ktorá simuluje šírenie akustických vĺn pomocou k-space pseudo-spektrálnej metódy. Táto aplikácia nachádza využitie hlavne v medicíne, kde umožňuje neinvazívne liečenie rakovinových nádorov pomocou ultrazvukového žiariča. Implementácia k-Wave skúmaná v mojej práci využíva rozdelenie globálnej simulačnej domény na lokálne sub-domény. Lokálna subdoména je potom spracovávaná na jednom GPU. Počas simulácie je však potrebné vymieňať medzi GPU okraje lokálnych sub-domén. Tento krok navyše ešte komplikuje fakt, že nie je možné prekryť prenos dát s výpočtom. Pre urýchlenie simulácie je preto potrebné zvýšiť priepustnosť dátových prenosov medzi GPU.

V práci je ďalej popísaná integrácia CUDA-Aware MPI do aplikácie k-Wave. Integrovaná bola aj podpora peer-to-peer prenosov pomocou CUDA Inter-process Communication (IPC). Tieto technológie umožňujú prenášať dáta medzi GPU priamo bez toho, aby boli pred tým kopírované do pamäte CPU. CUDA-Aware MPI a tak isto aj CUDA IPC preto dokážu simuláciu značne urýchliť v distribuovanom prostredí superpočítača a aj v rámci jedného uzla s viacerými GPU.

Výkon aplikácie k-Wave akcelerovanej pomocou CUDA-Aware MPI je ďalej analyzovaný na superpočítači Anselm v Ostrave, kde je 23 kusov Tesla K20m. Simulácia s podporou CUDA-Aware MPI tu dosahuje o 20-30 % kratší čas ako bez tejto podpory. To platí pre šírku prenášaných okrajov 16 bodov a rozloženie výpočtu medzi 16 GPU. Výkon je taktiež analyzovaný na serveri so 4 kusmi GTX 1080. V rámci jedného uzla je možné dosiahnuť o 10 % kratší čas simulácie pre použitie s dvomi GPU a až o 30 % pre 4 GPU pre veľkosť okraja 16 bodov. Na serveri s 8 grafickými kartami Tesla P40 môže byť čas simulácie až o 60 % kratší pri použití okraja veľkosti 16 bodov.

Efficient Communication in Multi-GPU Systems

Declaration

I hereby declare that this master thesis was prepared as an original author's work under the supervision of Ing. Filip Vaverka. The supplementary information was provided by doc. Ing. Jiří Jaroš, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Matej Špetko
July 31, 2018

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Filip Vaverka for his useful comments, remarks and engagement through the development process of this master thesis. Furthermore I would like to thank doc. Ing. Jiří Jaroš, Ph.D. for introducing me to the topic as well for the support on the way.

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“

Contents

Chapter 1

Introduction

After the introduction of Compute Unified Device Architecture (CUDA) by Nvidia, the graphics processing units (GPUs) shifted from devices used for rendering graphics into devices capable of accelerating any general purpose computation. GPUs are by design massively parallel processors. For this reason, some applications running on GPUs can achieve higher computation performance than running on general purpose CPUs. In addition, the power efficiency of the GPU is also better, when it achieves higher performance per Watt than a CPU. The scientific community noticed these advantages and GPUs became popular in high performance computing.

Many scientific applications require even more computation power than a single GPU can provide. Or the amount of data processed is larger than the capacity of its memory. For this reason the applications are scaled over multiple GPUs where the performance and efficiency of the computation is much higher compared to CPUs. In order to utilize multiple GPUs, some applications like password cracking do not require any data exchange. However, there are certain applications which need to exchange data during computation. This communication can represent a big overhead slowing down the overall performance of the GPU computation. As a result, it is important to research fast and efficient methods of data transfers between GPUs.

In this thesis, I examine methods of efficient GPU-to-GPU communication invloving lower latency, decreased CPU involvement, direct access to a GPU's memory and common host memory buffers and network controller. The methods researched include inter-node communication in a distributed supercomputer environment as well as intra-node: communication within a single server with multiple GPUs installed. The main focus is on GPUDirect technologies introduced by Nvidia and CUDA-Aware MPI.

Furthermore k-Wave toolbox for simulating the propagation of acoustic waves is introduced. Its multi-GPU implementation is accelerated using CUDA-Aware MPI. Peer-to-peer transfer capability is also integrated to this application by means of CUDA Inter-process Communication. The performance of accelerated version of k-Wave is analyzed and compared to the original version.

The following chapter describes the general purpose GPU computing. It explains why are graphic cards so popular among scientific community and why is it important to scale the computation over multiple GPUs. Furthermore, multi-GPU version of k-Wave toolbox is introduced and the reason for researching effective communication between GPUs is stated. Chapter 3 introduces the technologies enabling effective data transfers between GPUs. Chapter 4 describes the hardware I used for conducting my experiments. Chapter 5 describes the experiments conducted. It states how the bandwidth measurements were

performed, shows the results and discusses benefits of using the described technologies. Acceleration of multi-GPU implementation of k-Wave toolbox using CUDA-Aware MPI and peer-to-peer integration with CUDA Inter-process Communication is described in chapter 6. In chapter 7 the performance of accelerated versions of k-Wave toolbox is analyzed under a distributed environment. Chapter 8 analyses the performance on a single node. and chapter 10 contains the overall conclusion.

Chapter 2

Multi-GPU simulations (GPU computing)

In this chapter, I summarize the benefits of GPU computing. I explain the reasons why it is beneficial to use GPUs for general computations, why do scientific applications need to be scaled over multiple GPUs and why it is important to use effective communication methods in GPU-to-GPU communication.

2.1 GPU computing

Graphics processing unit – GPU was first developed as a dedicated hardware for accelerating rendering of graphics. Since the processing of graphics is inherently parallel problem, the GPU consists of many parallel units which can process lots of data in a single clock cycle. The later graphics applications required higher flexibility which resulted in creation of programmable GPUs. Since the GPU is basically a massively parallel processor the efforts were made to exploit its computational power for non-graphical applications. The concept of general purpose graphics processing unit (GPGPU) was born. Because of this researchers used this opportunity to utilize GPUs for many scientific applications.

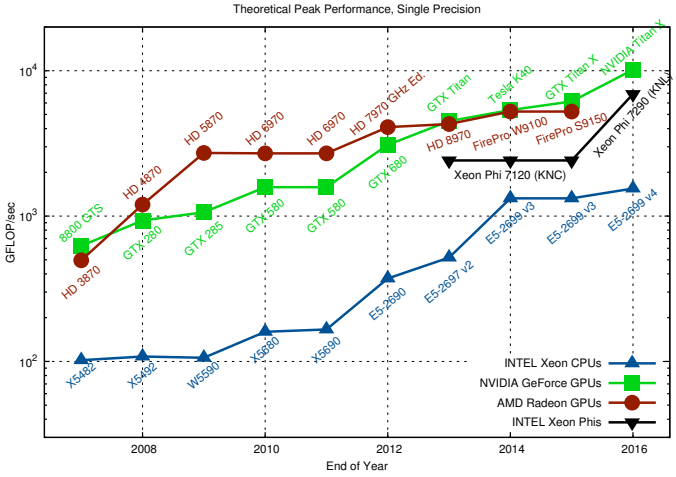


Figure 2.1: Maximum theoretical single-precision performance of CPU, GPU and MIC. (Image source [19])

Figure 2.1 compares the peak theoretical performance in single-precision of CPUs and GPUs over time. From the graph it is apparent that GPUs are in general 10 times more powerful. Moreover, the memory bandwidth is also important for the computational performance. Figure 2.2 shows that the latest GPUs can reach almost 1 TBps. However, the real heavily speedup depends on the application and it varies. In k-Wave's case GPU implementation is 8 times faster than optimized CPU version.[23]

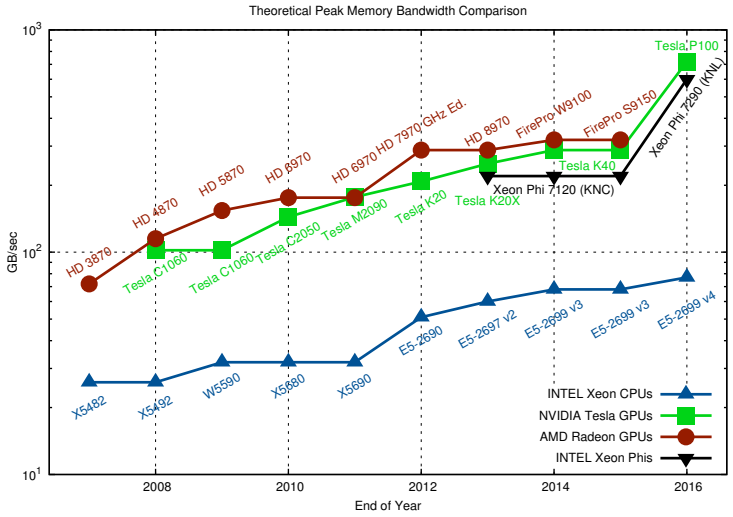


Figure 2.2: Comparison of theoretical peak memory bandwidth of CPU, GPU and MIC. (Image source [19])

2.2 multi-GPU

Because it is efficient to do scientific computation on GPUs, some of the most powerful supercomputers are equipped with a graphic accelerator. This can be seen in figure 2.3. There are two main reasons for scaling scientific applications over multiple GPUs. The first one is the amount of computation. If our application requires too much computation so that it takes too long time to finish. There might be a case when the result is required until certain deadline which can not be fulfilled by a single GPU.

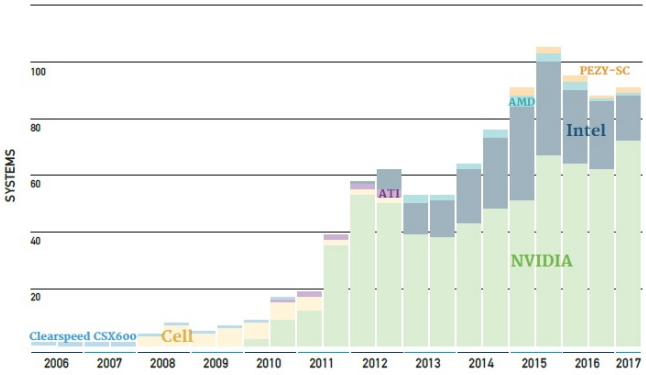


Figure 2.3: From TOP500 list, 91 supercomputers use accelerators. 71 of these machines use GPUs from Nvidia. (Image source [15][22])

The second reason is the amount of data being processed. GPUs have faster memory compared to the CPUs but it also has less capacity. The memory size is 4–8 GB for typical consumer class GPUs and 16–32 GB for high-end professional class GPU. However, if the application needs to process more than 512 GB of data, the GPU memory would serve as a substantial cache limited by the PCI-Express throughput.

2.3 k-Wave toolbox

The k-Wave is a open source tool for simulating the propagation of acoustic waves using k-space pseudo-spectral method. It is being developed by University College London and Brno University of Technology. The ultrasound simulations are mainly used in medicine. One of medical applications of k-Wave is planning HIFU treatment of cancer. HIFU – High-intensity focused ultrasound is a non-invasive therapy in which a tightly focused beam of ultrasound is used to rapidly heat tissue in a localized region until the cells are destroyed.[10] But in order to target the tumor it is necessary to properly align the ultrasound transducer. Knowledge about required setup is gained from simulating the propagation of ultrasound waves through the patient's body.

There are several versions of the toolbox available depending on the platform. Smaller simulation problems can run in Matlab or they can be accelerated on a GPU. One of the requirements is ability to run simulations with large domain sizes – $4096 \times 4096 \times 4096$ grid points. Not only the grid of these dimensions can not fit into memory of a single GPU, but it even can not fit into single node's host memory. It takes several days to run the simulation with CPU implementation utilizing hundreds of cores of a supercomputer. For this reason a version utilizing multiple GPUs was developed. The comparison for a grid size $1536 \times 1024 \times 2048$ points with 48 000 time steps along with the estimated simulation time for the current CPU and GPU implementation is shown at table 2.1.[11]

	Simulation Time	Simulation Cost
96 GPUs	14 h 9 m	475 $
128 GPUs	9 h 29 m	426 $
128 CPUs cores	6 d 18 h	1826 $
256 CPUs cores	3 d 0 h	1623 $
512 CPUs cores	2 d 5 h	2395 $

Table 2.1: Simulation time and cost when running a production simulation on Emerald with 96 and 128 GPUs, or Anselm with 128, 256, 512 CPU cores (Table source [11]).

In order to split the simulation domain among multiple GPUs the global simulation domain has to be decomposed. The multi-GPU version of k-Wave decomposes domain using local Fourier basis. What this means is basically that each GPU has its own local sub-domain where it performs computation. One simulation step consists of a local multiple FFTs computation. After this the overlapping regions – halo zones are exchanged between neighbours. The size of the overlapping region influences accuracy of the computation. This multi-domain decomposition – as it is also called – is shown in figure 2.4.[11]

Graph 2.5 shows the computation and communication time of the multi-GPU implementation of the k-Wave. The test was conducted on Emerald cluster for the domain size of $512 \times 512 \times 512$ and using 16 grid points overlap. We can see that the communication

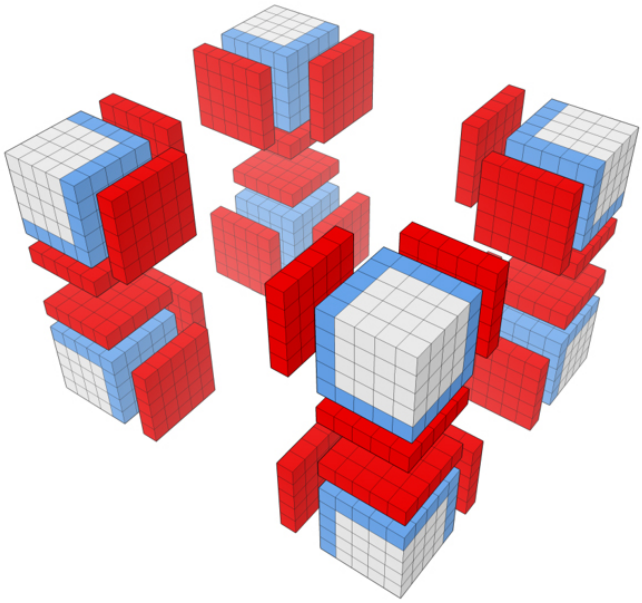


Figure 2.4: Multi-domain decomposition. Blue grid points are sent to neighbouring nodes and red are received. (Image source [23]).

time is over 50 % with the use of 16–32 GPUs. This represents a significant overhead of the simulation.[23]

This thesis investigates the efficient methods of multi-GPU communication in distributed GPU environments – cluster computers with 1 GPU accelerator per node. Furthermore the communication within single node with multiple GPUs is examined. The multi-GPU implementation of k-Wave toolbox as well as other scientific applications of which the performance is bound by communication can benefit from the methods described in this text.

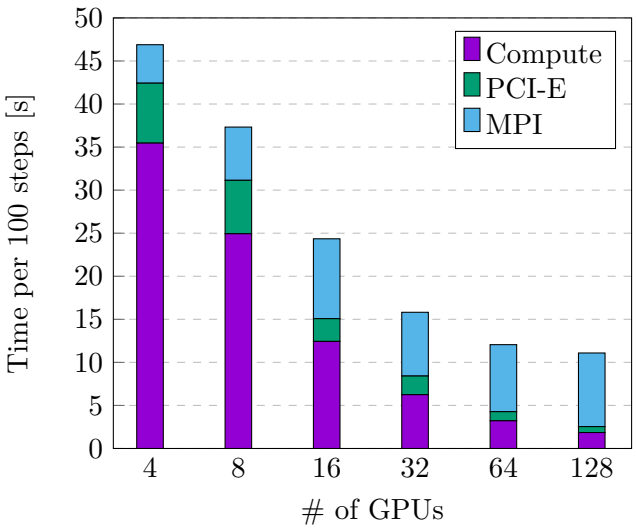


Figure 2.5: Simulation time for domain size 512^3 with 16 grid points overlap run on Emerald cluster. (Image source [23]).

Chapter 3

GPU communication methods

In this chapter I describe the key technologies enabling effective communication between GPUs. The technologies allow single node transfers as well as inter-node communication.

3.1 MPI

MPI stands for Message Passing Interface. It is a specification describing inter-process communication interface based on sending messages between processes.[14] This specification also includes support for dynamic process creation and parallel I/O operations. MPI is widely spread in distributed environments and it is supported by most of the supercomputers.

The MPI processes do not share memory with each other. The processes share their data via sending messages. The MPI message consists of a block of memory – an array containing serialized data. The supported data types are either standard ones (int, float) or more sophisticated data types can be defined with the MPI routines from the standard types including structures.

One of the advantages of the custom data types is that the library can partition the computation domain and spread these partitions among processes via optimized collective communication. For the use in distributed environment the MPI can ensure compatibility across different architectures. The library can convert data representation such as length of data types and endianness. Because of this it is safe to run computation on a cluster computer with nodes of different architecture.

3.2 GPUDirect technologies

GPUDirect is a series of technologies by NVIDIA introduced to simplify and accelerate data transfers between GPUs. With these technologies in use multiple GPUs, network adapters and other devices are able to read and write CUDA host and device memory directly. This eliminates unnecessary memory copies, reduces latencies and lowers CPU overhead. The result is significant performance improvement in data transfer times.

3.2.1 Unified Virtual Addressing – UVA

Unified Virtual Addressing was introduced in CUDA 4.0. This technology connects the host address space with the GPU address space into one large virtual address space. The memory space referenced by a pointer becomes transparent to application code. [2]

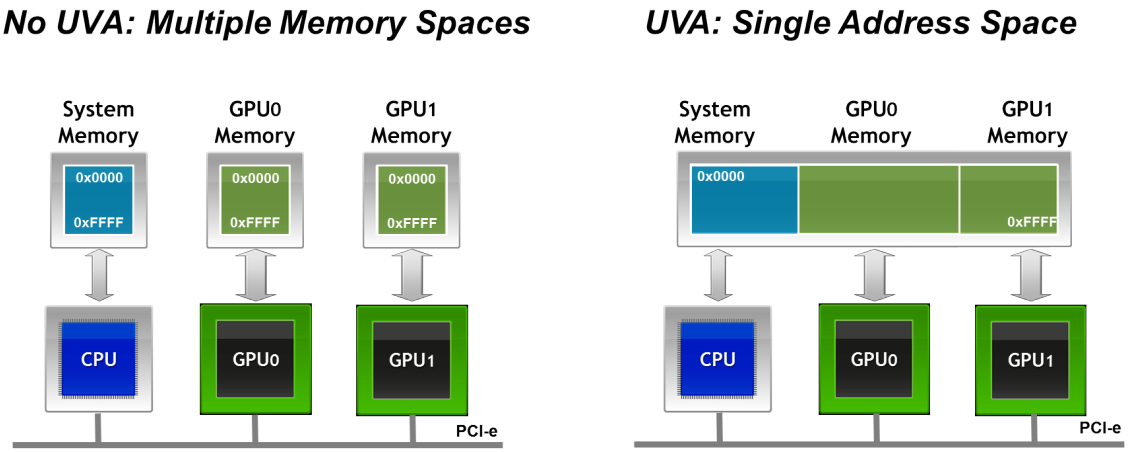


Figure 3.1: Host memory and device memory share a single virtual address space. (Image source [12]).

3.2.2 CUDA peer-to-peer

With the support of the P2P technology since CUDA 4.0 the intra-node transfers can be accelerated. Buffers can be directly copied between the memories of GPUs. This technology works only if the GPUs are connected to the same PCI-E root complex.

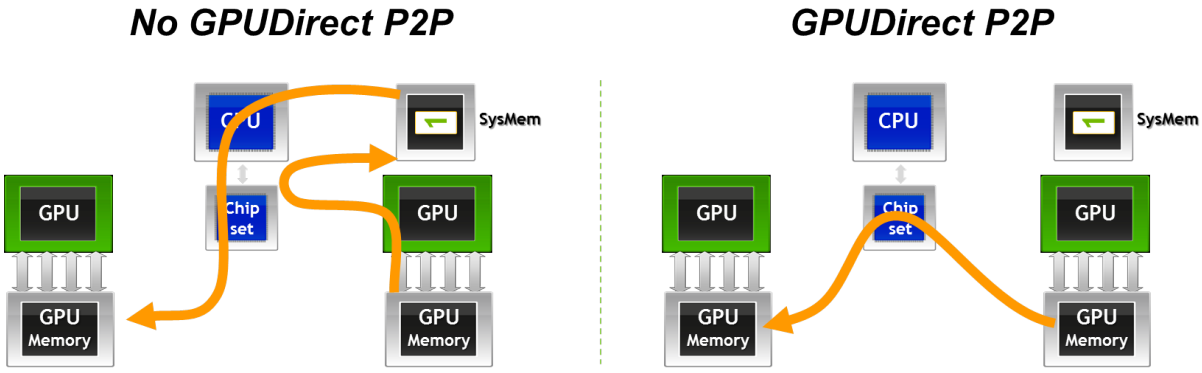


Figure 3.2: With the P2P access, the data can be transferred directly. (Image source [12]).

3.2.3 CUDA-Aware MPI

Several MPI implementations support this functionality, such as MVAPICH2, OpenMPI, CRAY MPI. MPI with CUDA aware support allows us to pass a pointer to a GPU memory directly to the MPI routine. This eliminates one copy to the host memory on a single node. This is shown in figure 3.3.

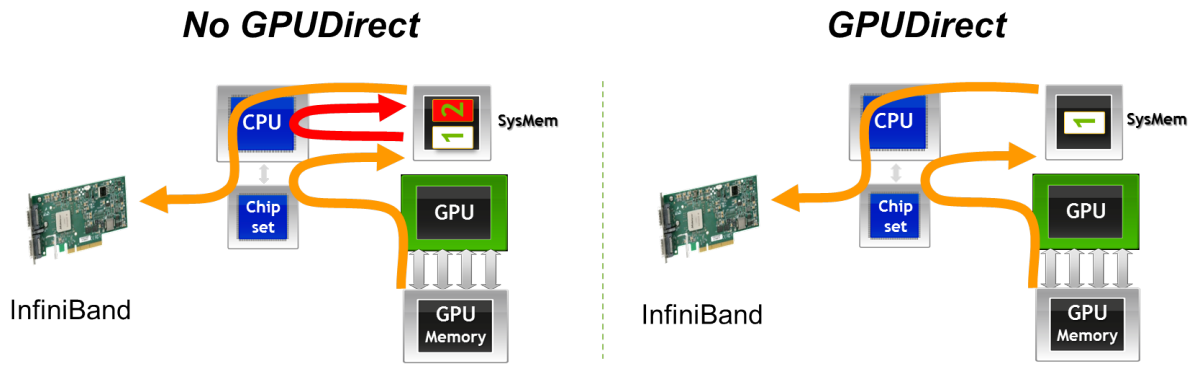


Figure 3.3: The use of CUDA-Aware MPI allows the MPI and CUDA to share a system buffer. (Image source [12]).

3.2.4 GPUDirect RDMA

RDMA stands for Remote Direct Memory Access. It allows the data to be passed to the network interface from GPU memory directly, without making a host copy. Unfortunately this only works with small messages $< 100\,\text{kB}$. This functionality is supported in CUDA-Aware MPI.

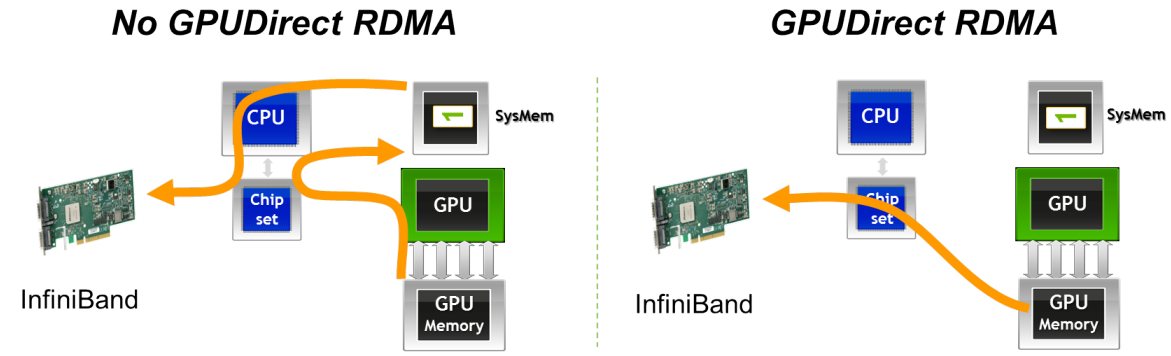


Figure 3.4: RDMA allows data to be transferred directly from the GPU memory to the network interface. (Image source [12]).

Chapter 4

Experimental hardware

This chapter describes the architectures of three machines used for testing technologies mentioned in the previous chapter. Also the key technologies influencing bandwidth of GPU communication are discussed. The first machine is Anselm supercomputer. Anselm provides 23 compute nodes equipped with GPU accelerator. These nodes are inter-connected with high throughput network – InfiniBand which makes this machine suitable for testing inter-node GPU-to-GPU communication. The second one is a single node server called SC-GPU1 run by Super Computing research group at FIT BUT (SC@FIT). This machine is not equipped with high throughput network but it has 4 GPUs installed and it is suitable for testing intra-node communication and peer-to-peer (P2P) transfers. The third machine is called Kinsler and it is run by the Biomedical Ultrasound Group at University College London. The machine is also a single node server equipped with 8 Tesla P40 GPUs. These are professional class GPUs and the machine is suitable for testing intra-node communication and P2P transfers.

4.1 Anselm

Anselm is the first of the two supercomputers administrated by National Supercomputing Center IT4Innovations at Technical University of Ostrava. Its purpose was focused to let Czech academic and research community get some experience with HPC and learn how to work with cluster computer. The second supercomputer run by IT4Innovations – Salomon is newer and more powerful. It is currently placed in the list of 500 most powerful supercomputers: TOP500 [22]. Unfortunately there are no GPU accelerators used on Salomon. Instead it is equipped with Many Integrated Core (MIC) accelerators Intel Xeon Phi. The topic researched in this thesis is relevant to all types of accelerators connected via PCI-E. However, main focus is on GPU communication. Therefore, the Anselm was chosen to conduct GPU-to-GPU data transfer experiments.[9]

Anselm consists of 209 compute nodes – 180 general purpose nodes without accelerator (1–180), 23 nodes equipped with GPU accelerator (181–203), 4 nodes equipped with MIC accelerator (204–207) and 2 fat compute nodes having 512 GB of memory (208–209). For my thesis the most important are the GPU accelerated nodes. Each of them has two Sandy Bridge Intel Xeon E5-2470 CPUs – 8 cores, 2,3 GHz clock speed and 20 MB of L3 cache. There is 96 GB of memory available in two NUMA nodes, 48 GB for each CPU.[20]

Nodes are supplied with NVIDIA Tesla K20 GPU accelerator. It is a professional class accelerator based on Kepler architecture. It has 2496 CUDA cores and 5 GB of memory

from which is only 4,5 GB available to user due to error correcting codes (ECC) turned on.[16] Tesla K20 is connected via PCI Express (PCI-E) bus version 2.0. Using 16 PCI-E lanes it can achieve up to 8 GBps peak theoretical throughput in each direction.[18]

All compute nodes of Anselm are interconnected by InfiniBand (IB) network using fully non-blocking fat-tree topology. The configuration used is 4 links Quad Data Rate with 40 Gbps throughput. After subtracting 8b/10b encoding overhead the effective peak theoretical limit is 4 GBps. This represents the upper limit of achievable throughput when it comes to GPU-to-GPU inter-node data transfers on Anselm. Moreover, achievable GPU-to-GPU data transfer latency is limited by the latency of InfiniBand.[6]

4.2 SC-GPU1

The machine used for testing P2P transfers between GPUs is called SC-GPU1. It is a research server of the HPC research group at FIT BUT – SC@FIT. SC-GPU1 is composed of two Haswell architecture Intel Xeon E5-2620 v3 CPUs running at 2,4 GHz. Each of them has 6 cores and 15 MB of L3 cache.[8] There is 32 GB of memory for each CPU, 64 GB in total.

SC-GPU1 is equipped with 4 NVIDIA GTX 1080 GPUs. This is a consumer type of GPU based on Pascal architecture. It has 2560 CUDA cores, 8 GB of memory and it is connected to host via PCI-E 3.0. [3] The difference between 2.0 and 3.0 version of PCI-E is that the version 3.0 increased signaling rate and switched from 8b/10b encoding to 128b/130b. This means that the peak theoretical throughput is almost doubled – 15,8 GBps.[18]

Motherboard Supermicro X10DRG-Q allows two GPUs to be connected to one CPU. Only this configuration is supported by the motherboard.[21] The GPU connection topology is shown at image 4.1. This represents a limitation concerning P2P transfers. P2P transfers are allowed only for devices within the same PCI root complex. In this case only GPUs connected to the same CPU are capable of P2P data transfers between each other. For transfers between GPUs connected to different CPUs the data have to be staged through the CPU memory and pass through the QPI link. This is the main bottleneck when performing GPU-to-GPU intra-node transfers on SC-GPU1.

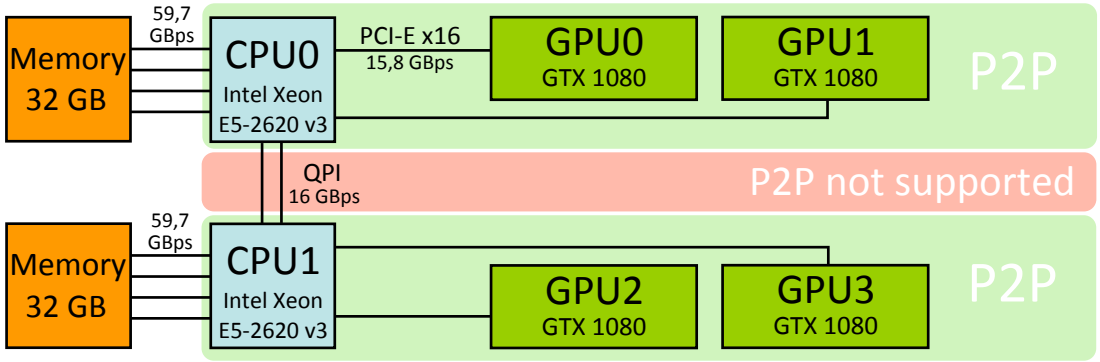


Figure 4.1: Block diagram of GPU connection topology on SC-GPU1. (Image source [1, 21, 8])

4.3 Kinsler

Another machine used for testing intra-node P2P communication – Kinsler. It is run by Biomedical Ultrasound Group at University College London. Kinsler has two Broadwell architecture Intel Xeon E5-2620 v4 CPUs running at 2,1 GHz. Each of these CPUs has 8 cores and 20 MB of L3 cache.[7] There is 256 GB of memory for each CPU, 512 GB in total. Kinsler is equipped with 8 NVIDIA Tesla P40 GPUs. It is a professional type of GPU based on Pascal architecture. It has 3840 CUDA cores, 24 GB of memory. GPUs are connected to host over PCI-E 3.0.[17]

Motherboard used in this 8–GPU server is Gigabyte G250–G52. For each CPU there are only 2 PCI-E slots with 16 PCI-E links. In order to connect 4 GPUs to a single CPU, a pair of GPUs is first connected to a PCI-E switch. Then the two PCI-E switches are connected to the CPU directly. This enables P2P access between 4 GPUs connected to single PCI-E root complex represented by the CPU. GPU connection topology is shown in figure 4.2. [5]

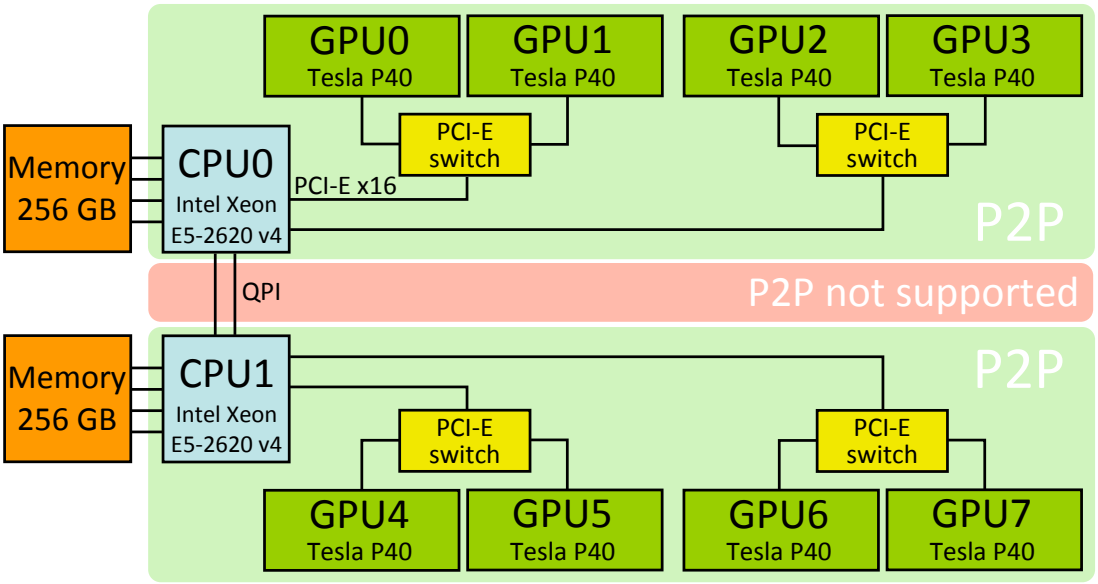


Figure 4.2: Block diagram of GPU connection topology on Kinsler. (Image source [5, 7])

Chapter 5

Communication benchmarking

This chapter outlines the practical impact of using described communication methods on the experimental hardware. The goal of my experiments is to compare the bandwidths of a GPU-to-GPU data transfer using a CUDA-Aware MPI implementation and regular MPI. I will describe a way how I conducted the data transfer rate measurements and report the results. These benchmarks were inspired by CUDA Samples and OSU Micro-Benchmarks.

Currently there is no CUDA-Aware MPI implementation installed on Anselm. To be able to conduct my experiments I had to build my own CUDA-Aware MPI module. For building the module I used EasyBuild framework. This framework is used in the HPC environments to install and manage scientific software in an efficient way.[4] The easyconfig file for installing CUDA-Aware OpenMPI library was present at the IT4I's easyconfig repository. On SC-GPU1 machine the OpenMPI library was already installed with CUDA-Aware support.

5.1 Simple bandwidth test

First test is focused on measuring bandwidth of data transfer between GPUs. It is a simple ping-pong data exchange between the memories of the two GPUs. The transfer rate is measured using regular OpenMPI library and compared to CUDA-Aware MPI implementation. Regular OpenMPI uses `cudaMemcpy()` call to copy data from GPU memory to host memory and then the data are passed to `MPI_Send()` call. In CUDA-Aware MPI version the `cudaMemcpy()` call is left out and a pointer to GPU memory is passed to MPI call directly.

Many scientific applications including k-Wave with multi-domain decomposition need to do the data exchange. Because of this the data transfer is tested in both directions simultaneously and a non-blocking `MPI_Isend`/`MPI_Irecv` routines are used. Both GPUs have allocated memory buffers for sending and receiving data. The data are exchanged 200 times and average throughput is reported.

5.1.1 Anselm MPI

Compute nodes used for this test are cn201 and cn202 connected to the same InfiniBand switch. MPI library used is OpenMPI 2.0.2. As CUDA-Aware library, the same version was used only built with CUDA-Aware support. Version of CUDA used is 8.0.61. For comparison a host-to-host data exchange is also measured. Figure 5.1 shows the results of the measurement.

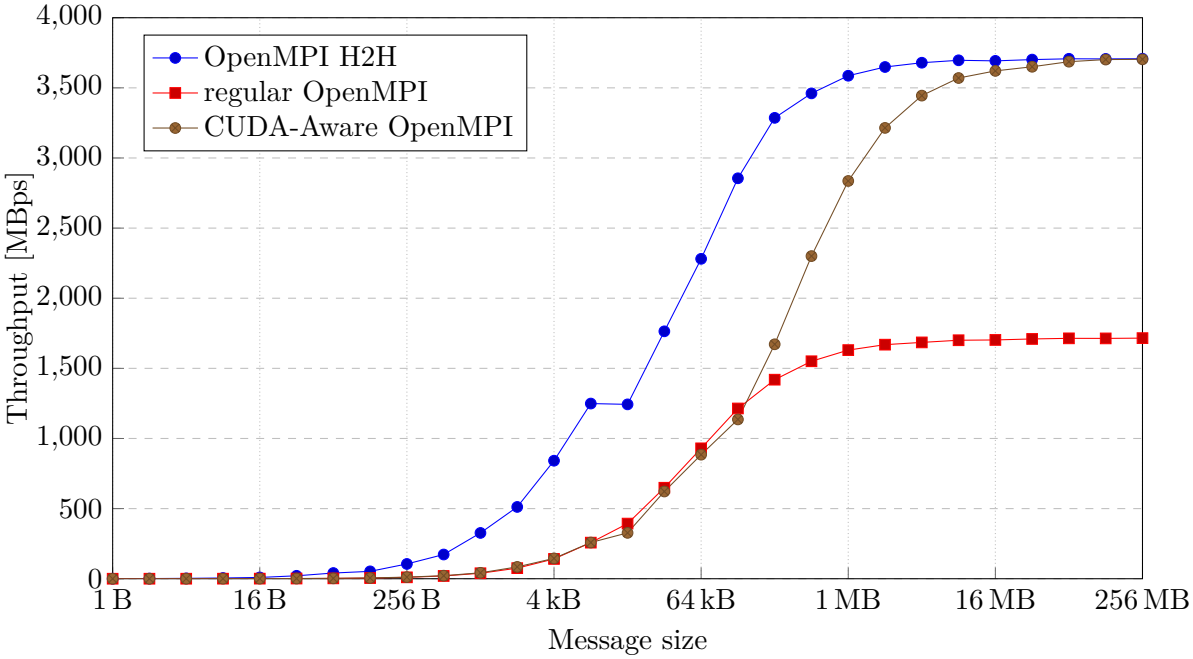


Figure 5.1: Comparing regular MPI and CUDA-Aware MPI bandwidths of a GPU-to-GPU bidirectional data transfer on Anselm. Message size doubles every step. Blue line represents Host-to-Host transfer.

The way CUDA-Aware MPI works is that it has a common system memory buffer for outgoing/incoming MPI message and `cudamemcpy()` function. This means that before message reaches the destination GPU memory, it is staged through host memory twice – first time at sender and second time at receiver. Regular MPI does not have these common buffers. Instead it makes two extra copies through host memory. In total, the message is staged 4 times. This can be seen in figure 5.1: the throughput of regular MPI (red) is approximately half of the CUDA-Aware MPI (brown) throughput for message sizes $\geq$ 4 MB.

CUDA-Aware MPI reaches the highest performance for message sizes 64 MB and higher. The achieved bandwidth is around 3 700 MBps. For message sizes higher than 256 MB the bandwidth does not get any bigger – peak throughput of the InfiniBand interconnect was reached. The latency for small messages ($<$ 4 kB) is around 22 μs – the throughput for these messages gets close to zero MBps. This is also caused by the host staging. The host-to-host transfers have much lower latencies – one magnitude lower. Because of this the throughput curve of host-to-host transfers rises earlier.

If the RDMA technology was used, the latency for small messages of the GPU-to-GPU transfer would be much lower – the transfer rates would be higher and the curve would go closer to the host-to-host curve. Unfortunately RDMA is not available at Anselm cluster, so there is no way for me to test its influence. Figure 5.2 shows the throughput using RDMA. This the throughput was measured by Jiri Kraus and Peter Messmer and it was taken from [13].

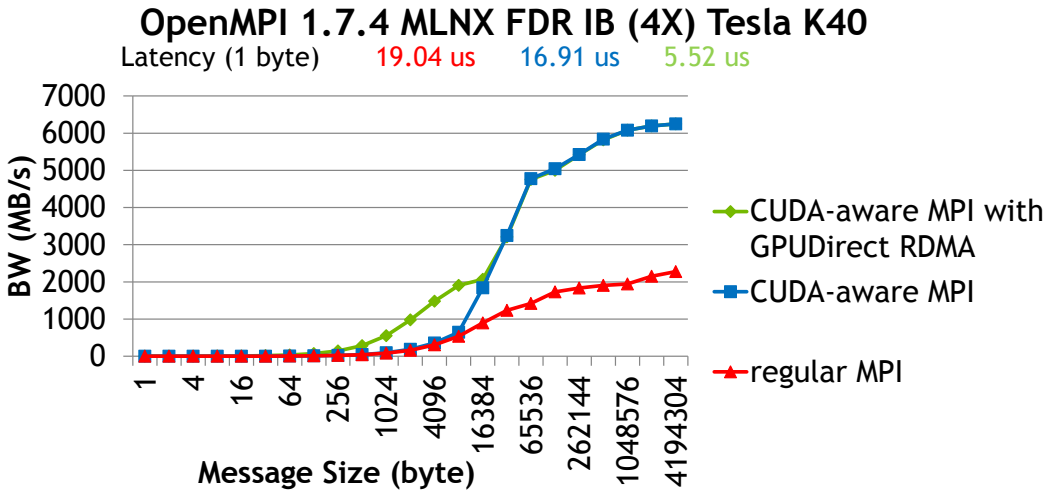


Figure 5.2: The throughput of CUDA-Aware MPI RDMA (green line) transferring small messages. (Image source [13]).

5.1.2 SC-GPU1 P2P

The method of this test is the same as in the previous case except that no MPI is involved. Data transfers are controlled from one thread only. Function `cudaMemcpyPeerAsync()` manages the data transfers. In order to test transfer rate in both directions, two instances of this call are issued simultaneously in two different CUDA streams. The function also matches the memory pointer with correct GPU. Before the transfer starts, the P2P access has to be turned on by `cudaDeviceEnablePeerAccess()` within both GPUs. Switching between active GPU is done by `cudaSetDevice()` function.

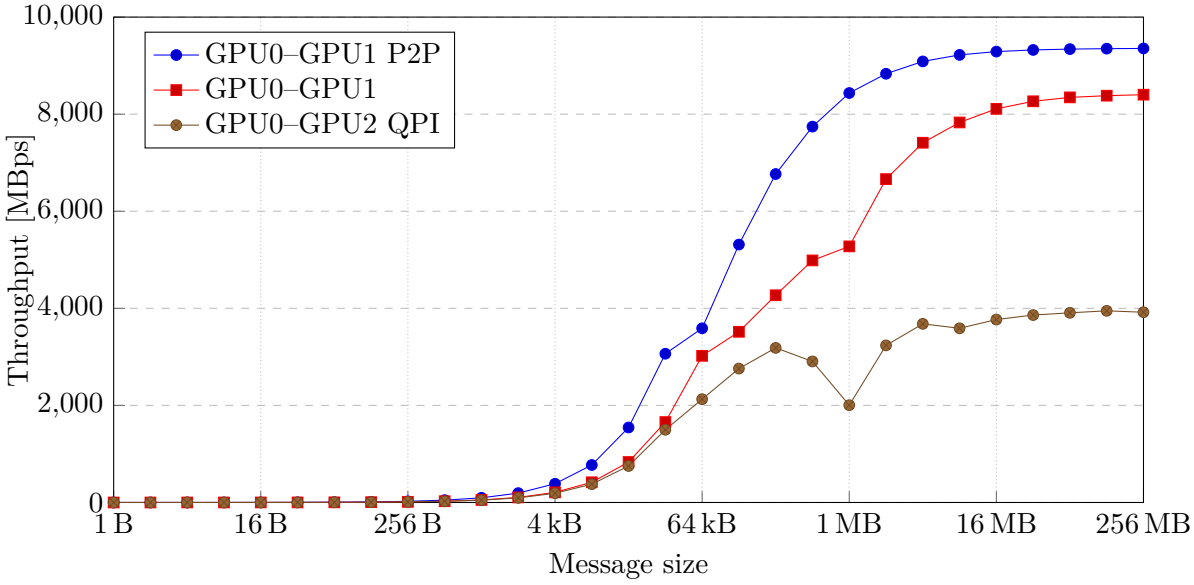


Figure 5.3: Bandwidth of a GPU-to-GPU bidirectional data transfer on SC-GPU1 managed by CUDA. The blue and red lines show transfers within the same PCI root complex with P2P access enabled and disabled. The brown line displays inter-socket transfer over QPI link. Message size doubles every step.

P2P transfers are available only between GPU0-GPU1 and GPU2-GPU3 as it is shown in figure 4.1 in the previous chapter. I measured the bandwidth between GPU0-GPU1 two times: with P2P access turned on and off. The third test performed data transfer between GPU0-GPU2 over QPI link. CUDA version 8.0.61 is being used to perform these experiments. The measurements were done 400 times to hide warm up latencies and the average results are presented.

Figure 5.3 shows results of this test. The peak bandwidth of a P2P transfer – 9 350 MBps was achieved for the messages $\geq$ 8 MB. With the P2P access turned off, the highest transfer rate is 1 GBps lower – 8 400 MBps achieved for messages $\geq$ 32 MB. For the inter-socket transfers (GPU0–GPU2) the peak bandwidth reached is 3 900 MBps. This indicates that transfers over QPI link have limited transfer rate because the CUDA copy function used in this experiment stages data through the host memory at least two times.

The latency of small messages $\leq$ 32 kB for P2P transfer is around 10 μs. Transfer with disabled P2P access has this latency around 19 μs and for the transfer over QPI link it is 22 μs.

5.1.3 SC-GPU1 MPI

This test was conducted in the same way as the test performed on Anselm, except this time the MPI is used within single node. The goal of this test is to examine whether the CUDA-Aware MPI implementation installed on SC-GPU1 can benefit from the intra-node P2P access. CUDA GPUDirect technologies, especially CUDA UVA should allow MPI to utilize P2P transfers. Data transfers within single CPU socket (GPU0–GPU1) and across both sockets through QPI (GPU0–GPU2) are analysed. The transfers were repeated 400 times. To perform this test CUDA version 8.0.61 and OpenMPI version 2.1.1 is being used.

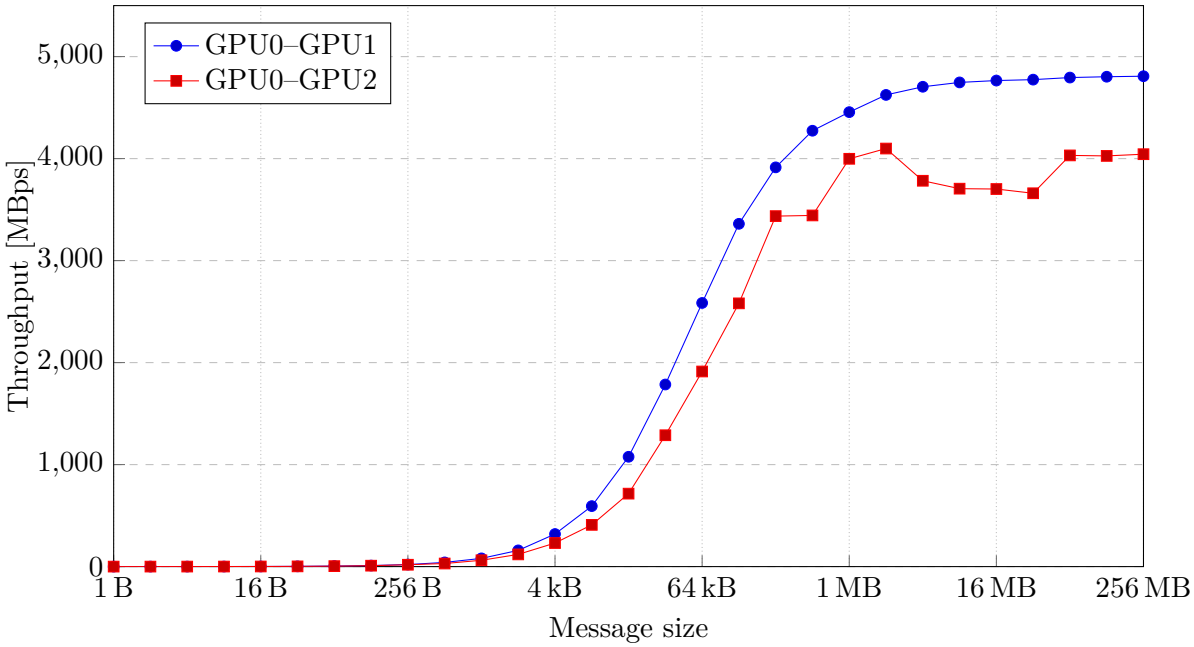


Figure 5.4: Bandwidth of a GPU-to-GPU bidirectional data transfer on SC-GPU1 handled by MPI. The blue line shows transfer between devices within the same PCI root complex. The red line displays inter-socket transfer over QPI link. Message size doubles every step.

Figure 5.4 shows that the bandwidth of the MPI is approximately half of the CUDA P2P bandwidth. The peak transfer rate for the 8 MB message and higher is only 4 750 MBps. The latency for messages $\leq$ 8 kB is 12 μs. The transfers over QPI reach the peak bandwidth at 4 000 MBps for message sizes $\geq$ 8 MB. The QPI transfers have latency 14 μs but only for messages under 512 B.

MPI parameter `btl_smcuda_cuda_max_send_size` was set to 64 MB. This parameter sets the size in bytes for the largest message that can be transferred from GPU to other GPU via shared memory. The default value is set to 128 kB. When the default value is used data transferred from and to GPUs are split into 128 kB chunks and the overall throughput is limited to approximately 3 150 MBps. With message size also shared memory buffer size needs to be increased. This is done by setting another MPI variable – `btl_smcuda_min_size`. For measurements I used 4 GB. This shared memory buffer is used by MPI processes for copying data from one GPU to another in single node envirnoment.

These parameters are not be found in OpenMPI documentation. They were found by accident while I was researching poor data transfer performance on SC-GPU1. CUDA-Aware MPI can take advantage of P2P connection by using CUDA IPC. But it only reaches half of the bi-directional throughput of P2P transfer.

Data that go from GPU to GPU over QPI are transferred between MPI processes via shared memory buffer. These transfers are also conducted in one direction at a time. This behaviour might be because CUDA-Aware MPI uses only one CUDA stream to perform P2P transfers via CUDA IPC and another single stream for transferring data between GPUs over shared memory.

5.1.4 Anselm MPI all-to-all

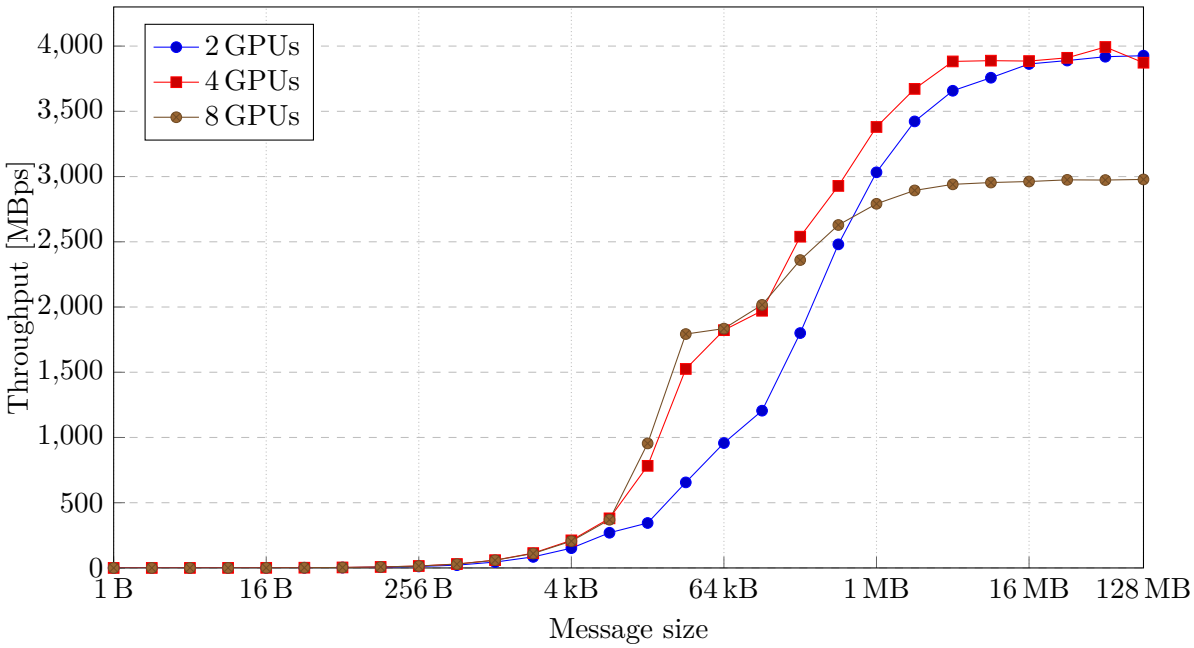


Figure 5.5: Bandwidth of a all-to-all data exchange between GPUs on Anselm. The transfer speed decreased by 1 GBps when switching from 4 GPUs to 8 GPUs.

This test does not perform true MPI all-to-all exchange. It is implemented using regular non-blocking send and recv calls. The reason for this is that each rank sends also data to itself which I consider redundant when operating with GPU memory. Each process sends message to every other process and it also receives message from every other process.

This test was performed on compute nodes connected to the same IB switch. Figure 5.5 shows the throughput of all-to-all data exchange for 2, 4 and 8 GPUs. But when conducted by 8 GPUs a transfer speed drop by 1 GBps can be observed. This might be caused by the saturation of IB interconnect.

5.1.5 Kinsler MPI all-to-all

This test is done in similar way to the previous one. But it is performed on a single node machine capable of doing P2P transfers within 4 GPUs. CUDA Inter-process Communication was used to enable P2P.

Unfortunately CUDA-Aware MPI parameter setting of maximum message size and shared memory buffer were not set when performing these measurements. This means that the 8 GPU performance is limited and it could be at least 1 GBps higher.

Figure 5.6 shows the performance of all-to-all data exchange on Kinsler. Comparing 2 GPU and 4 GPU throughput a significant bandwidth drop is noticeable. This might be caused by the architecture of the server and PCI-E interconnect saturation.

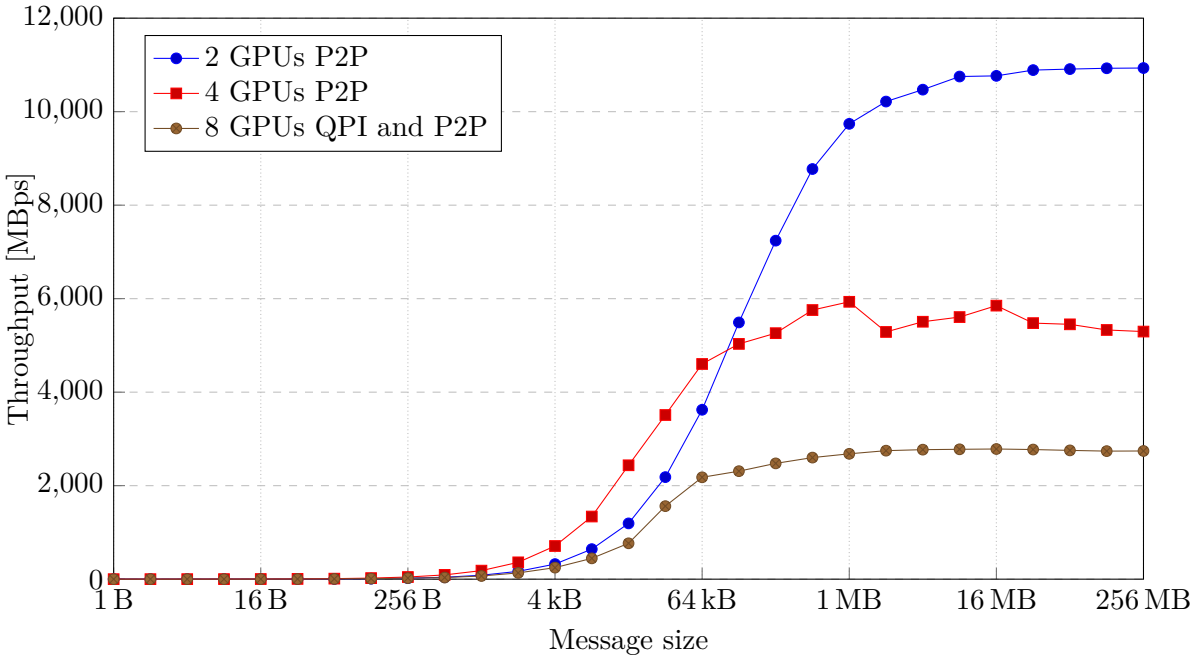


Figure 5.6: Bandwidth of all-to-all bidirectional data transfer on Kinsler.

5.2 Border exchange test

Several scientific applications work in the way that in order to split their computation to multiple computational nodes the global simulation grid is split into multiple local subdomains. Every computation node has its own local grid. After computing one simulation step the borders of the local grid have to be exchanged with neighbouring nodes. This

domain decomposition of a 3D grid is shown in figure 2.4. The situation is even worse for k-Wave since borders of several local grids need to be exchanged during one simulation step.

To be able to test the performance of 3D grid border exchange I developed a simple benchmark application. This application does not perform any computation, it only simulates the border exchange in a 3D grid. The amount of compute nodes and the grid size in each dimension can be chosen. A Cartesian communicator and custom MPI data types are used to simplify the global grid partitioning and scattering data to MPI ranks. Thanks to the Cartesian communicator MPI rank knows what are its neighbours and in which direction they are located. This information is necessary to send each border to its receiving rank.

The 3D grid's data are stored in GPU memory as 1D array. This means that the border data are not contiguous and they are spread through the whole array. Before exchanging borders the required data need to be extracted and serialized. My original intention was to use custom MPI data types types to serialize border data from the grid. This approach can work for data residing in the host memory where MPI packing routines are optimized. But to be able to extract data from GPU memory the needed memory region is transferred to host memory first and then the packing routines serialize data. This represents significant overhead as large amount of data is transferred over PCI-E bus and this approach can not be used.

To be able to exchange data as quick as possible the borders need to be extracted inside fast GPU memory. The sending and receiving buffers need to be allocated on GPU memory. To serialize border data a series of packing kernels is executed and the necessary data are copied to the buffers. The buffers are then sent to the receiver directly from GPU memory with an MPI call. Because non-blocking MPI send/receive functions are used a separate buffers for sending and receiving borders need to be allocated. This means that some extra space needs to be used in GPU memory.

This test was conducted on Anselm with 8 compute nodes. The grid size of the test was $768 \times 768 \times 768$ points, border width 2–32 grid points and data were exchanged 10 000 times so the start up latencies are hidden. In this test the borders are exchanged only with direct neighbours, so one node can perform at most 6 border exchanges. The node configurations for the test were: 2 GPUs with grid partitioned in x dimension, 4 GPUs with grid partitioned in x and y dimensions, 8 GPUs with grid partitioned in x and y and z dimensions.

The demonstration results of this application are shown in figure 7.5a. The overhead of packing kernels is less than 1% of the total transfer time. The amount of communication differs for every configuration. When the grid is partitioned equally in each dimension the amount of transferred data is less than for configurations partitioned unequally in each dimension. When the grid is partitioned in single dimension only, the node in the middle has to transfer more data than node on the edge. Overall performance is also higher for equally partitioned configurations. Figure 5.8 shows bandwidths for each configuration. Bandwidths are calculated from the node which transfers the most data in current configuration.

5.3 Performance evaluation

The results of my tests show that it is beneficial to use CUDA-Aware MPI library when we want to perform inter-node GPU-to-GPU data exchange. Compared to use of cudamemcpy

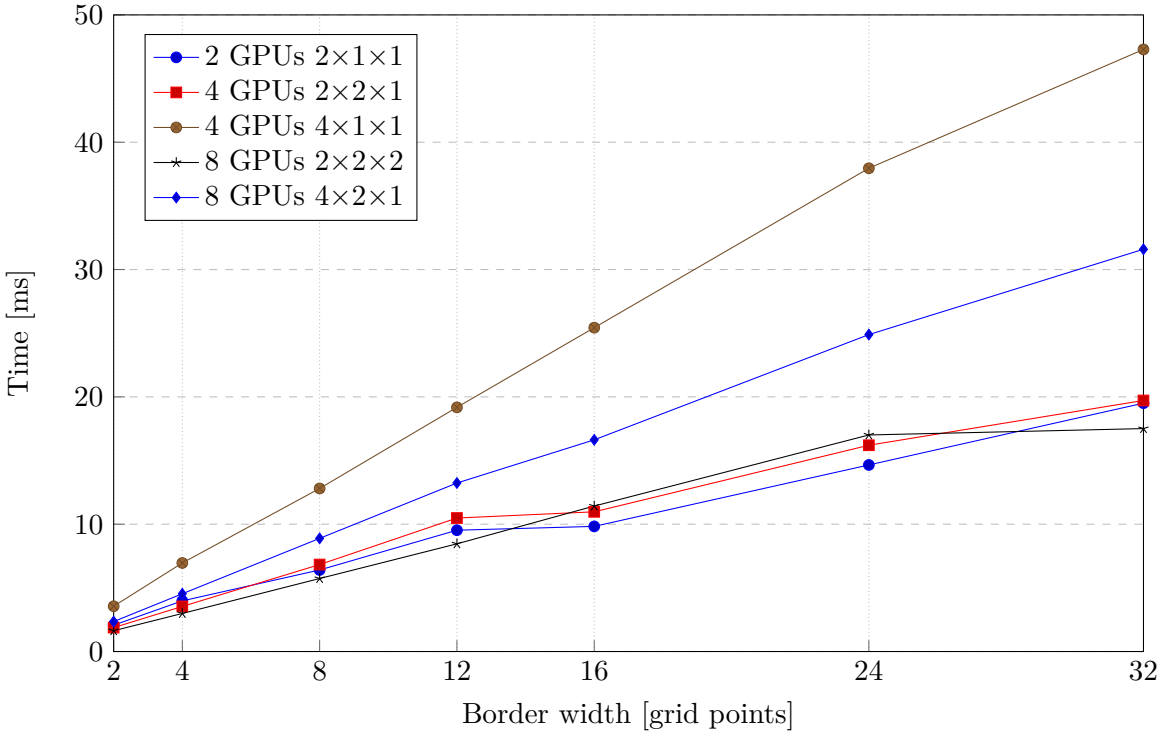


Figure 5.7: The average time of one border exchange

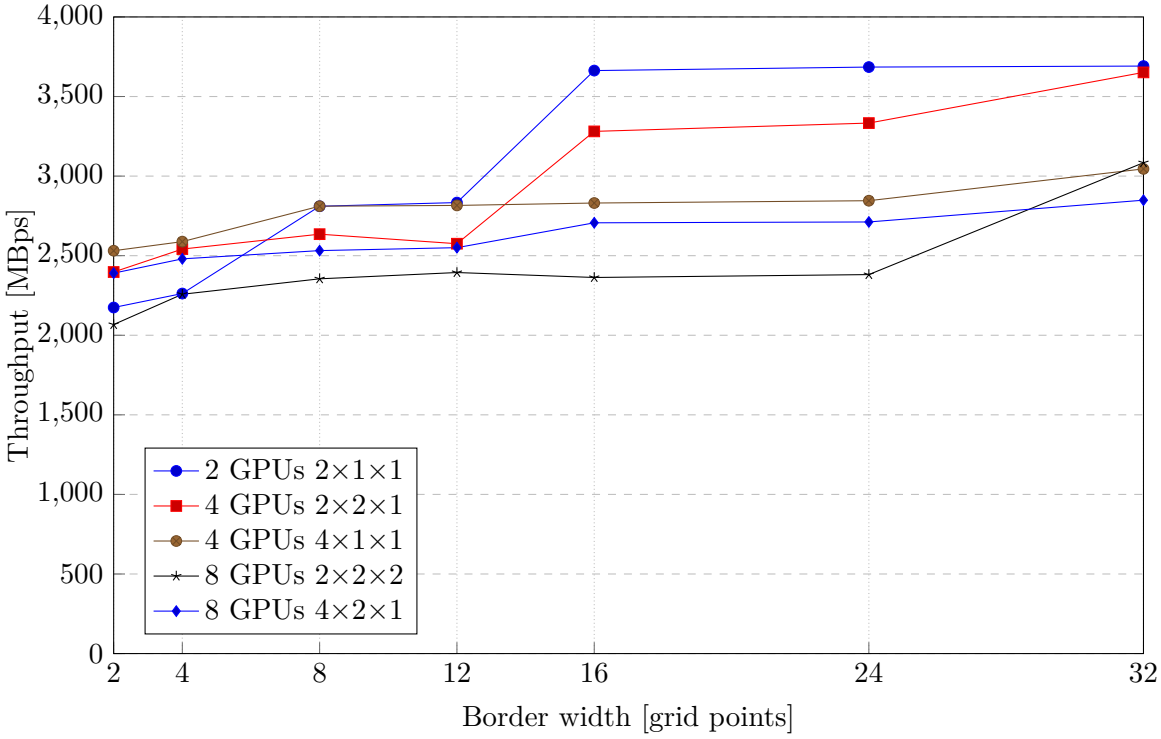


Figure 5.8: Throughput reached in the border exchange test.

and regular MPI, there is less overhead by two host memory copies and also the peak throughput of InfiniBand or other interconnect can be achieved. RDMA is advised to use when there is a need to transfer small messages $< 100\,\text{kB}$. Because the data are sent to the network interface directly from GPU memory it can significantly lower the latency of these messages. Bigger messages are staged through host memory. Unfortunately I was not able to test the influence of this technology because it is not supported on hardware where the experiments were conducted.

As for the intra-node transfers the bandwidth of a P2P data transfer was lower than the peak theoretical throughput of PCI-E bus. This might be because the PCI-E data transfers have higher overhead compared to InfiniBand interconnect. The tested version of MPI library was not able to utilize full throughput when P2P access is available. CUDA data transfer routines provided higher throughput than the MPI in this case. The MPI should be able to take advantage of the P2P transfers but this was not confirmed on the experimental hardware. If the highest intra-node throughput is required, an architecture that allows all the GPUs to be connected to the same PCI-E switch should be chosen. That way all the GPUs would have P2P access among each other and the highest bandwidth would be achieved.

Next chapter explains the integration of CUDA-Aware MPI into k-Wave application. P2P is also integrated to this application using CUDA Inter-process Communication.

Chapter 6

Data movement in k-Wave

This chapter describes the computation principle of the k-Wave toolbox. It also explained how are overlapping data transferred between processes during computation. Afterwards, the integration of CUDA-Aware MPI into k-Wave toolbox is described. Subsequently, P2P integration using CUDA Inter-process Communication (IPC) is explained. The first part of next section was taken from [11].

6.1 k-Wave fluid Local Domain Decomposition

K-Wave is an advanced acoustic wave propagation simulation framework based on the k-space pseudo-spectral method. Acoustic wave propagation in homogeneous media can be expressed by a set of coupled first-order partial differential equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho_0}\nabla p + S_F \tag{6.1}$$

$$\frac{\partial \rho}{\partial t} = -\rho_0 \nabla \cdot \mathbf{u} + S_M \tag{6.2}$$

$$p = c_0^2 \rho \tag{6.3}$$

Here $\mathbf{u}$ is the acoustic particle velocity, p is the acoustic pressure, c_0 is the sound speed, and ρ_0 and ρ are the ambient and acoustic density, respectively. These equations are solved using the k-space pseudo-spectral method, where spatial gradients are computed using the Fourier collocation spectral method, and time integration is performed using a dispersion-corrected finite difference scheme. [11]

$$\frac{\partial}{\partial \xi} p^n = \mathcal{F}^{-1}\left\{ ik_\xi \kappa e^{ik_\xi \Delta\xi/2} \mathcal{F}\left\{p^n\right\}\right\} \tag{6.4}$$

The equation 6.4 represents a single component of gradient p from equation 6.1. Function $\frac{\partial p}{\partial \xi}$ is derivative of p with respect to ξ which determines the dimension in 3D space. This derivation is computed by Fourier transform of the function p multiplied by $ik_\xi \kappa e^{ik_\xi \Delta\xi/2}$, where i is imaginary unit, k is wave number, κ represents correction coefficient and exponent of $e^{ik_\xi \Delta\xi/2}$ moves the function of half of grid-point. Multiplying each Fourier coefficient with its corresponding wave number realizes differentiation by shifting each complex exponential appropriately. Finally, derivative of p with respect to ξ is computed by taking inverse FT. Parameter n represents number of the current simulation time step.

$$u_\xi^{n+\frac{1}{2}} = u_\xi^{n-\frac{1}{2}} - \frac{\Delta t}{\rho_0}\frac{\partial}{\partial \xi} p^n + \Delta t S_{F_\xi}^n \tag{6.5}$$

The equation 6.5 is based on equation 6.1. It is a function of integration $\frac{\partial u}{\partial t}$ over one time step. This equation is in the form of the Euler method, where u in the next half simulation step is computed as u from last simulation half step plus gradient p and ultrasound source S_F. Equation 6.4 steps into this equation as gradient p. Parameter ξ determines the same dimension in 3D space as in equation 6.4. Δt represents a time step of the simulation.

As mentioned in section 2.3, several different implementations of k-Wave toolbox exist. In this thesis, I will focus on fluid simulation which uses local domain decomposition. This implementation can be run either with OpenMP threads, CUDA or OpenCL. My thesis describes data transfer optimization of CUDA backend.

In this implementation, the global simulation domain is partitioned into smaller local domains. Each local domain is assigned to single MPI process. MPI process than manages computation on a single GPU. Parallel I/O is used to load input data from HDF5 input data file and also to store output result. The most difficult part of the computation is calculating FFTs using cuFFT 3D.

When global grid is partitioned in 3 dimensions, each MPI process has to exchange border data with its 26 neighbours. Even when the process has no neighbour in a particular direction, it exchanges overlapping data with the process on the opposite end of the grid. This is because the FFT has a periodic nature and it is important for FFTs to maintain this periodicity in local domains as well as in global domain. Another thing caused by the periodicity is that after exchanging overlapping regions a bell function needs to be applied to them. It is necessary for points located at the start of the local domain to have the same value as points at the end. Therefore, points closest to the edges of local domain are smoothed to zero by the bell function.

The width of overlapping edges influences precision of the overall computation. The most commonly used border width is 16 points because it makes a good compromise between precision and the amount of transferred data. But minimal sufficient accuracy is achieved by using 8 points wide overlap.

Figure 6.1 shows simplified dataflow model of one simulation step. Orange boxes represent data exchange. The computation and communication can overlap only partially because FFTs require to have all overlapping data exchanged before they start to compute. The overlap can happen only after `matrix_1` finished border exchange and started computation. In the meantime `matrix_2` and `matrix_3` can exchange borders. But in other cases, the computation and communication is not possible and therefore it is important to optimize communication.

6.2 Simulation data exchange

Several local matrices storing data are required for k-Wave computation. This section describes data exchange of a single matrix happening during one simulation step. The data exchange between MPI processes is managed by communication framework. It is responsible for partitioning global simulation domain into local sub-domains and also for the exchange of overlapping regions within neighbouring processes.

At the initiation phase, an MPI Cartesian communicator is used to create the desired grid of processes. It determines the process's position in the grid. It is also used for assigning neighbour processes to each direction. The Cartesian topology is periodic – this means that the last process in a certain dimension is neighbour to the first process in that dimension and vice versa. Whether the grid is partitioned in 1D, 2D or 3D each process has 2, 8 or 26 neighbours respectively. Knowing the position in Cartesian topology the dimensions of

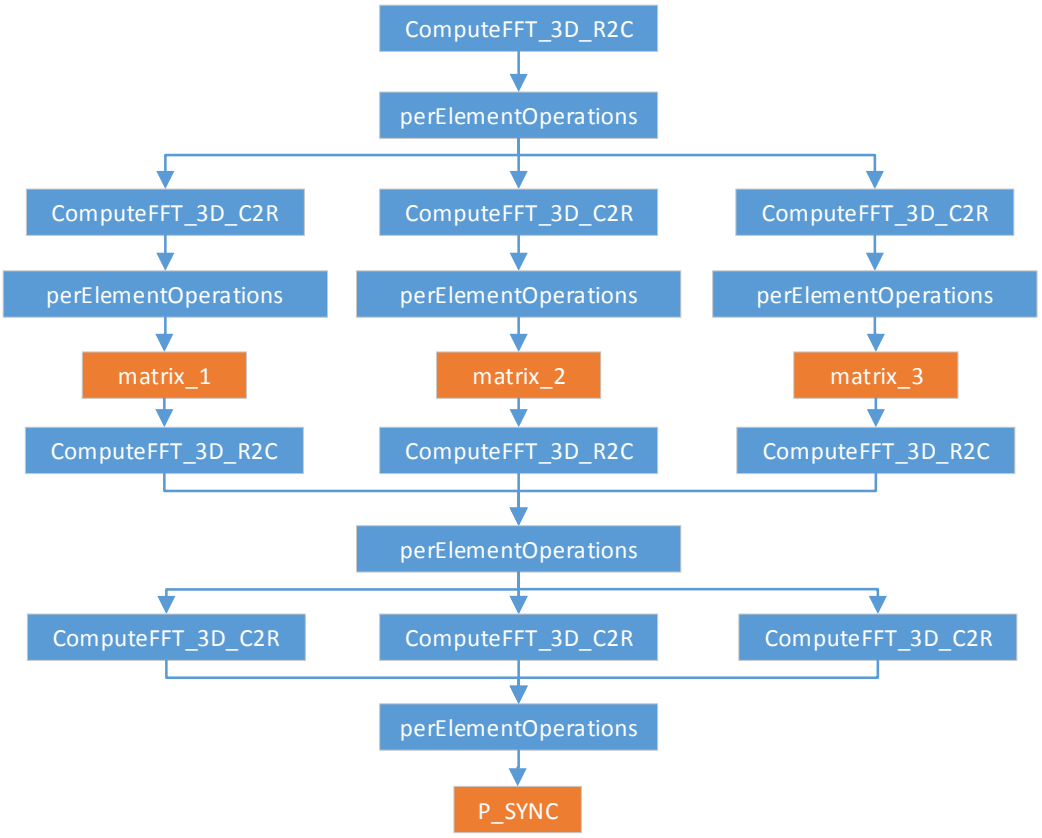


Figure 6.1: Simplified data flow diagram of k-Wave fluid LDD one simulation step. Orange boxes represent data exchanges.

a local matrix and offsets from the global matrix are calculated. In the next step memory space is allocated in CPU and GPU address spaces for the local matrix and also for send and receive buffers. The appropriate data are then loaded into the local matrix from HDF5 file including overlapping regions.

Subsequently, persistent exchanges, so-called links, are initiated with each neighbouring rank. Each link represents one part of the local matrix to be exchanged between two neighbour processes: border or corner. The link contains target rank ID, pointers to send and receive buffers in CPU, offset from the buffer start where the exchange data is stored in buffers and a custom MPI datatype which represents the length of data to be exchanged. When links to all neighbour nodes are created, two arrays of persistent communication requests are initiated. The send requests array is created with `MPI_Send_init()` and the receive requests array with `MPI_Recv_init()` calls.

After the initiation phase, the procedure to exchange overlapping data during computation is as follows: first, CUDA extraction kernels are issued to copy border data from matrix to send buffer. Since the matrix is stored as a 3D array, the border points are spread across whole memory block where the matrix is stored. This means that the data points of each border need to be serialized into the send buffer. The advantage of this data serialization is that this buffer is copied from GPU memory to CPU memory in a single `cudaMemcpy()` call. Another advantage of serialization is that the matrix data can be used for other computation after the data are copied to separate buffer.

When the data is copied to the CPU send buffer, the actual data exchange is started. This is accomplished by non-blocking MPI point-to-point communication. The communication is issued by two `MPI_Startall()` calls which start all the persistent exchanges saved in the arrays of send and receive requests created in the initiation phase. Afterwards, `MPI_Waitall` is used to wait until all the send and receive requests finish and received data are safely stored in the CPU receive buffer. Data is then copied from the receive buffer in CPU memory space to receive buffer in GPU memory with another `cudaMemcpy()` call.

With the data copied in GPU receive buffer, CUDA injection kernels are launched. These kernels do the opposite operation to extraction kernels – they de-serialize data from receive buffer and copy it into the matrix. While the data are injected into the matrix, the Bell function is applied to them.

The whole data exchange process of a single MPI rank is illustrated in figure 6.2. In k-Wave this exchange is done in 3D grid but to simplify, the illustration shows border exchange for 2D grid. The important thing to notice is that not only borders are stored in send buffer, but the corner points are stored as well and they need to be stored separately from the borders because they are sent to different MPI processes.

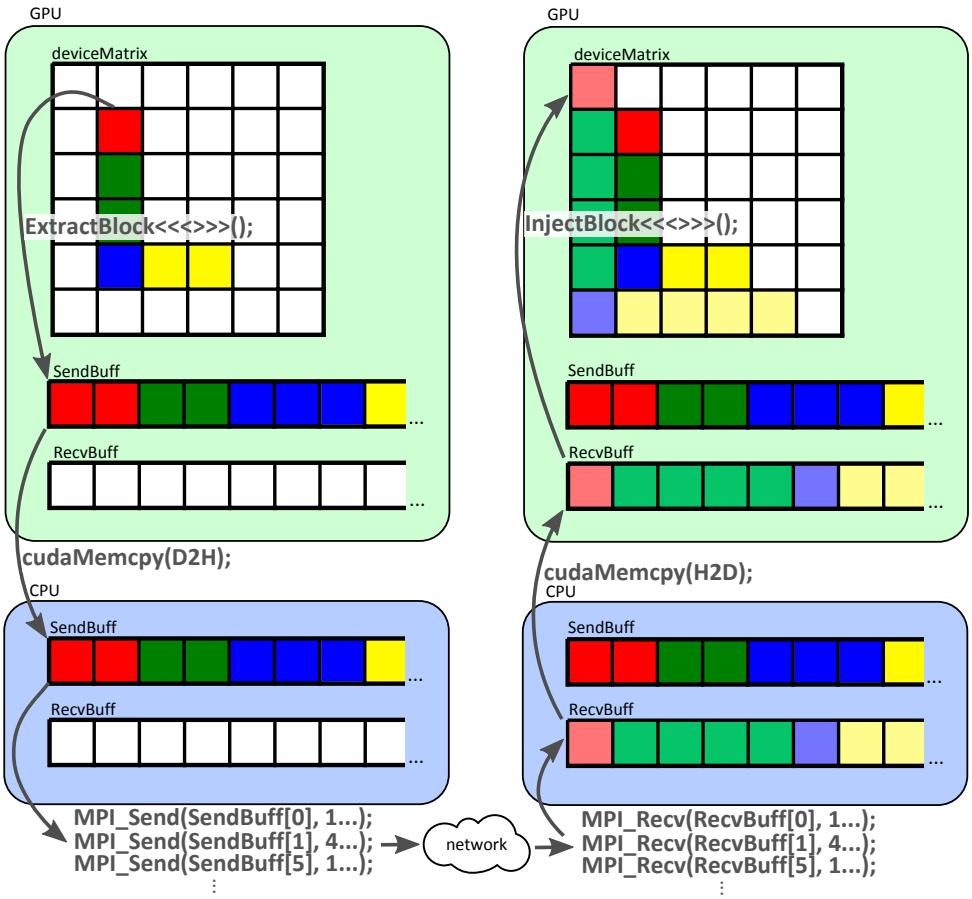


Figure 6.2: Illustration of data exchange in k-Wave for single MPI process. To simplify, data exchange is shown for 2D grid but in k-Wave this is done in 3D grid. The important thing is that the red point is stored in the send buffer twice. The first time it is stored as a corner point and the second time as a part of 4 point left border. Similarly, the blue point is stored three times: twice as a part of two borders and once as a corner point. In the receive buffer, the borders and corner points are displayed separately.

6.3 CUDA-Aware MPI integration

The main advantage of CUDA-Aware MPI is that it can access GPU memory directly. There is no need to conduct device to host transfer before sending data or host to device after receiving data over MPI. This section describes the modifications done to the k-Wave fluid LDD application in order to use acceleration by CUDA-Aware MPI.

Two changes were made in the data exchange of k-Wave fluid LDD in order to integrate CUDA-Aware MPI. Firstly, it was necessary to omit device to host copy of send buffer and also host to device copy of receive buffer. These memory copies are not needed because CUDA-Aware MPI can access this data directly from GPU memory. Second, change was made to the setup of persistent MPI data exchanges. These data exchanges – links perform the actual data exchange between MPI processes. These links are responsible for transferring data from senders send buffer in CPU to receivers receive buffer also in CPU. The CPU buffers are copies of GPU buffers so the source and destination pointers in MPI links can be changed from CPU buffers to GPU buffers. There is no need to copy data from and into CPU buffers when GPU buffers can be accessed directly from MPI.

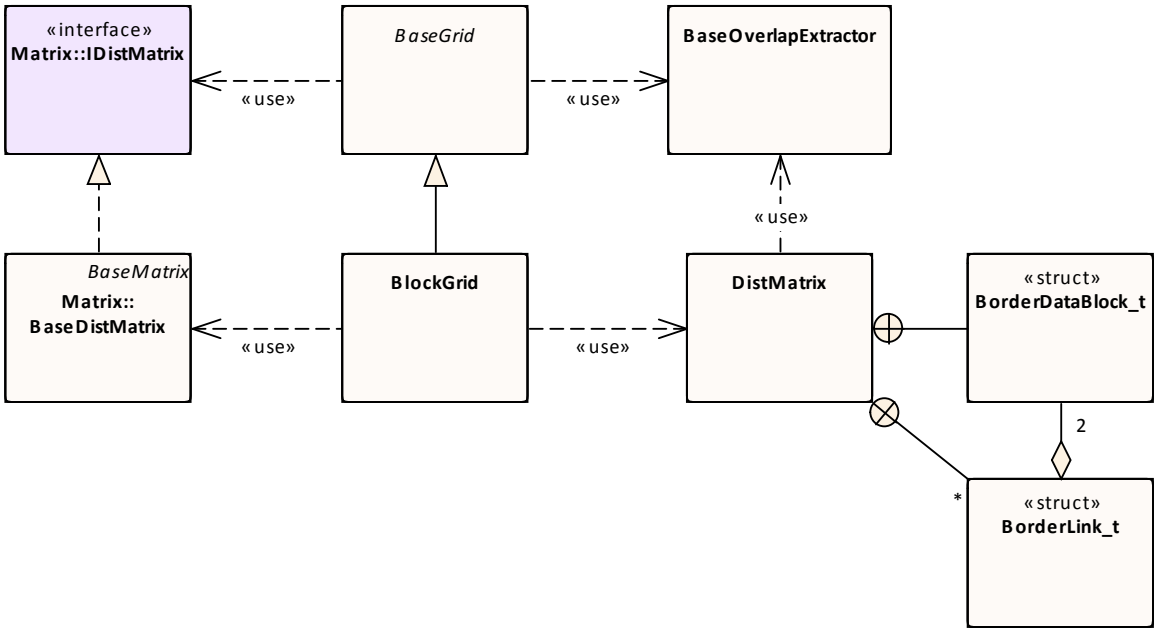


Figure 6.3: Class diagram of k-Wave communication framework. (Image source: k-Wave Fluid LDD documentation)

Modifications were made to the communication framework module `comm_fabric` and to `OverlapExtractor` class in `backend_cuda` module. Important classes for modification of `comm_fabric` module are `DistMatrix`, `BlockGrid` and `BaseGrid`. `DistMatrix` contains information about local matrix as well as send and receive buffers for exchanging borders. Pointers to these buffers are stored in structure `BorderDataBlock_t`. The `BlockGrid` class creates all the links with neighbouring processes and stores each link in `BorderLink_t` structure. `BaseGrid` is responsible for creating arrays of MPI send and receive requests. The relationships in the `comm_fabric` module is shown in figure 6.3.

Function `CreateLink` implemented in class `BaseGrid` takes `DistMatrix` as a parameter. It calculates all the necessary parameters of one particular data exchange link. That

includes target rank and the length of data that are about to be exchanged. It also includes an address from which are the data about to be copied when sending and where are they going to be stored when receiving. This address consists of a pointer to start of send/receive buffer and an offset representing where is the particular border block located in the buffer. This offset is the same for send and receive buffer. The whole border pointer for send data block is stored in `pSendData` value and for receive in `pRecvData`.

To use CUDA-Aware MPI the pointer to the send/receive buffer needs to be replaced. Instead of using buffer in CPU memory, it is possible to copy data directly from GPU buffer. The offset stays the same because the GPU buffer contains exactly the same data as CPU buffer. The CPU buffer pointer is accessed from `DistMatrix` through function `GetRecvDataBlock()` or `GetSendDataBlock()`. This function returns `BorderDataBlock` containing property `pBlockData`. This property holds pointers to the CPU and GPU buffer. The CPU buffer is accessible with a `GetRawData()` call and similarly the GPU buffer can be accessed by calling `GetDeviceData()`. Property `pSendData` and `pRecvData` contain pointers to GPU buffer and they are later used in `BaseGrid` class to create arrays of persistent communication requests.

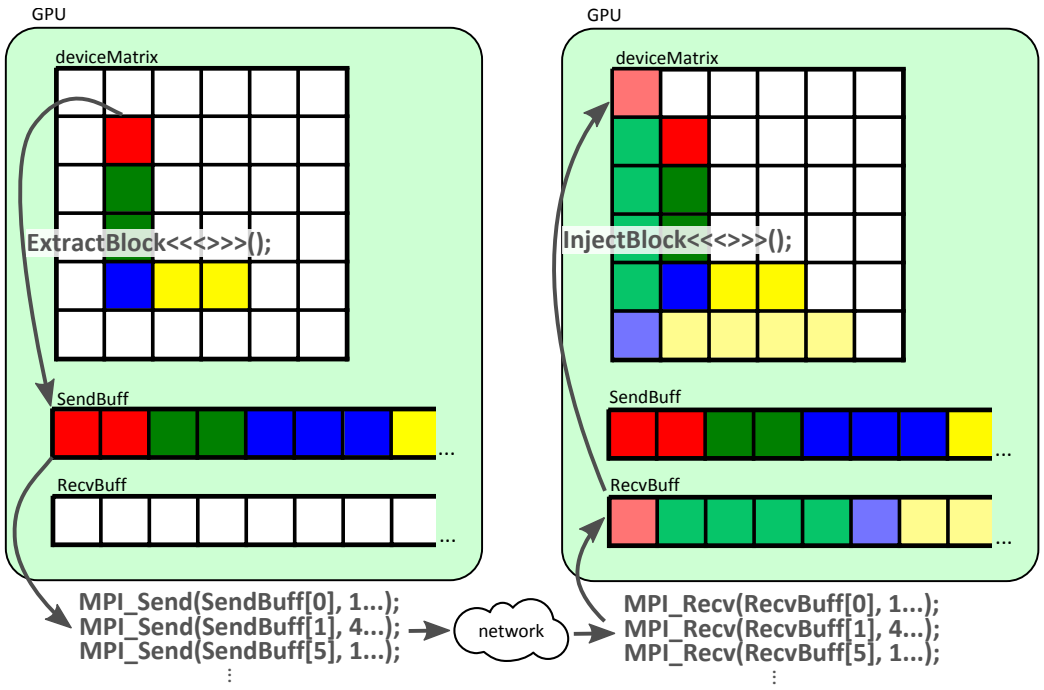


Figure 6.4: Illustration of data exchange in k-Wave for single MPI process using CUDA-Aware MPI. To simplify, data exchange is shown for a 2D grid but in k-Wave this is done using a 3D grid. The data exchange is similar to original version shown in figure 6.2. The difference is that data are copied directly from buffers allocated in GPU.

The second modification was made to the `OverlapExtractor` class implemented in `backend_cuda` module. This class contains function `ExtractOverlaps()` which takes care of serializing border data from local matrix into GPU buffer. Additionally, it copies the data from GPU buffer to CPU buffer. On the other hand, function `InjectOverlaps()` copies data from CPU buffer to GPU buffer. Afterwards, it deserializes them and saves them into the local matrix. These device to host and host to device copies can be removed from these functions since CUDA-Aware MPI can access data in GPU buffer directly.

Also, additional synchronization with default computation stream was added into the `ExtractOverlaps()` function. The synchronization was added after the extraction kernels are started. The issue was that data were sent to receiving process before they were extracted to the send buffer. Because of this, the receiving process received wrong data. This additional synchronization ensures that the data are sent after they are correctly extracted. Figure 6.4 shows data exchange for one MPI process after the integration of CUDA-Aware MPI.

6.4 P2P integration with CUDA IPC

In normal conditions, the integration of P2P support to k-Wave fluid LDD would include switching from MPI processes which control each GPU, to OpenMP threads. The advantage of using threads is that they have a common address space. Therefore, any thread can access memory of any GPU controlled by another thread. Additionally, the `cudaMemcpyPeer()` function can be used. This type of `cudaMemcpy` uses P2P transfer where possible. If data need to be copied between CPU sockets (over QPI), it stages them over CPU memory automatically. Unfortunately, this approach can be used only in a single node environment.

The current implementation of k-Wave is bound to MPI processes because it uses parallel I/O, specifically parallel implementation of HDF5 library to load and store data files. Because of this, the approach mentioned in previous paragraph can not be used. The main issue is that GPU memory allocated in an MPI process can not be seen by other processes. The only way to integrate P2P capability is with the use of CUDA Inter-process Communication (IPC). CUDA IPC allows to export GPU pointer from address space of one process into an address space of another process. The second process can work with this memory pointer as if it was local.

To setup an IPC communication link, a GPU memory pointer needs to be converted to a `cudaIpcMemHandle_t` structure. This memory handle structure is created from a local GPU pointer by `cudaIpcGetMemHandle()` function. In the case of k-Wave, the memory pointer points to the receive buffer in GPU. Afterwards, all-to-all exchange of the IPC memory handle is performed within every MPI process. Subsequently, each MPI process tries to convert every received IPC memory handle back to GPU pointer. The conversion is made by `cudaIpcOpenMemHandle()`. The conversion is successful when there is P2P connection available between these GPUs.

This way, the MPI process can find out which other processes are accessible by P2P. If there is no peer access, the conversion of IPC memory handle fails and no memory pointer is returned by `cudaIpcOpenMemHandle()` function. Instead, it will set an error indicating no available peer access. This error needs to be reset because it is propagates later in the application. This method of determining peer accessibility works also in distributed environment. If `cudaIpcOpenMemHandle()` encounters IPC memory handle from another node, it is not converted.

The data exchange in k-Wave uses persistent exchanges – MPI links. When performing these exchanges, the sending process does not know the offset the receiving process uses to store the data in its receive buffer. To perform data exchange over IPC, `cudaMemcpy()` function is used. As the type of transfer, `cudaMemcpyDefault` is set. More importantly, this function needs to know the receiving process' offset in order to place the data to the correct location in the receive buffer. Therefore, this offset value needs to be sent from receiving process to sending process. The exchange of the receiving offset is the last thing needed to set up an IPC link. When the data are exchanged between MPI processes, CUDA IPC link

is used if P2P access is available for these processes. If P2P is not available regular MPI link is used utilizing CUDA-Aware MPI. Regular MPI links are not influenced by CUDA links. This is why this approach can work also in distributed environment.

To enable computation and communication overlap, asynchronous type of memcpy is used. The `cudaMemcpyAsync()` needs to specify a stream. For this reason a stream dedicated to data exchange is created with `cudaStreamNonBlocking` flag. This flag ensures that the stream does not synchronize with the default stream. Only one copy stream is created in a single MPI process as this is sufficient. For each local matrix in MPI process an event is created with two flags: `cudaEventBlockingSync` and `cudaEventDisableTiming`. The first flag ensures that the host is blocked during synchronization and the second one improves performance since it is not necessary to store timing data.

The processes need to know when is the asynchronous copy finished. After all the asynchronous copies are started an event recorded into the copy stream. When the process synchronizes to this event, it is blocked until all the copies are finished. CUDA IPC alows to export event as well. This event could be exported from sending process to receiving process so that the receiving process can synchronize to it. But synchronizing may cause a problem. The problem is that the receiving process can synchronize to the event before it is recorded by sending process. The event is in set as an already occured and the receiving process would continue without the correct data.

Because of this problem the synchronization is handled in different way. The sending process is only synchronized to its local event. When the copy is finished, one byte long MPI message is sent to all the receiving processes. This message is intended to signals all the receiving processes that the copy was finished.

CUDA IPC in this implementation works only with CUDA-Aware MPI support. Because there is no point in transferring some data over CPU buffer and some data directly from GPU. With CUDA-Aware MPI all the data are transferred directly from GPU. And for any reason it can be turned off during compilation.

Chapter 7

Performance in distributed environment

In this chapter, the performance measurements of k-Wave fluid LDD application is described. These measurements were conducted on Anselm supercomputer in distributed environment. The performance of CUDA-Aware MPI version of k-Wave is evaluated and compared to original version which uses regular MPI.

7.1 Simulation parameters

Performance was measured in distributed environment on Anselm cluster, where every node contains a single GPU. Input files were generated with `p0_source_input` parameter meaning that only initial pressure is set and no other pressure sources are applied during simulation. Domain sizes used in experiments were in range of $256^3 - 512 \times 1024 \times 1024$ doubling one dimension size with each run. Overlapping region was set to $2 - 32$ grid points, also doubling with each run. Because there are only 23 nodes with GPUs on Anselm the simulation was run with $1 - 16$ GPUs. Domain decompositions used for measurements are as follows (x, y, z): 1 GPU – 1, 1, 1; 2 GPUs – 1, 1, 2; 4 GPUs – 1, 2, 2; 8 GPUs – 2, 2, 2 and 16 GPUs – 2, 2, 4.

K-Wave has an integrated profiler which was used to collect precise timing data using `MPI_Wtime()`. Time of CUDA operations is measured by synchronisation and not by CUDA events. Profiler reports these timings for each MPI process independently. In my measurements I use an average value from all the processes. To measure only computation and communication performance, output data were not sampled during simulation. Only final pressure state in the whole domain is saved and this step is not counted in measurement. For each run 100 simulation steps were performed to hide any warm up latencies.

7.2 Strong scaling

Figures 7.1 – 7.5 show strong scaling of original version compared to CUDA-Aware MPI version. To create these graphs the mean value of single simulation step time was taken from k-wave profiler reports of all the measurements.

These measurements show that the scaling starts to improve from 8 GPUs to 16 and for 2, 4, and 8 GPUs is not as good. This is due to the grid partitioning between GPUs. When the grid partitions in another dimension the amount of communication between nodes

raises. Going from 8 to 16 GPUs the domain is already partitioned in all 3 dimensions so the amount of communication does not raise significantly.

Another thing worth mentioning can be seen when comparing results having border width 2 in figure 7.2 with border width 4 in figure 7.3. Some cases of simulation having border width 2 points perform worse than the once with 4. The example of this case is domain size $2^8 \times 2^9 \times 2^9$ for 2 GPUs. Simulation with border width 2 performs worse because its local domain size has higher prime factor. cuFFT library is optimized only for certain prime factors. The higher the prime factor, the worse is its performance. The edge size in x dimension is 260 for the case with border size 2. 260 has the highest prime factor 13. Compared to the case with border size 4 when the edge size is 264 and the prime factor 11.

Variable time of computation caused by prime factors within different border widths can be observed in most of the measurements. This is not that important since my thesis is not focused on optimizing computation. It is focused on optimizing data transfers and this does not have influence to them.

Figure 7.8 shows the execution of CUDA-Aware MPI version relative to original version. It shows $20-30$ percent faster execution time of CUDA-Aware version compared to original version. This aplies for border width 16 and 16 GPUs

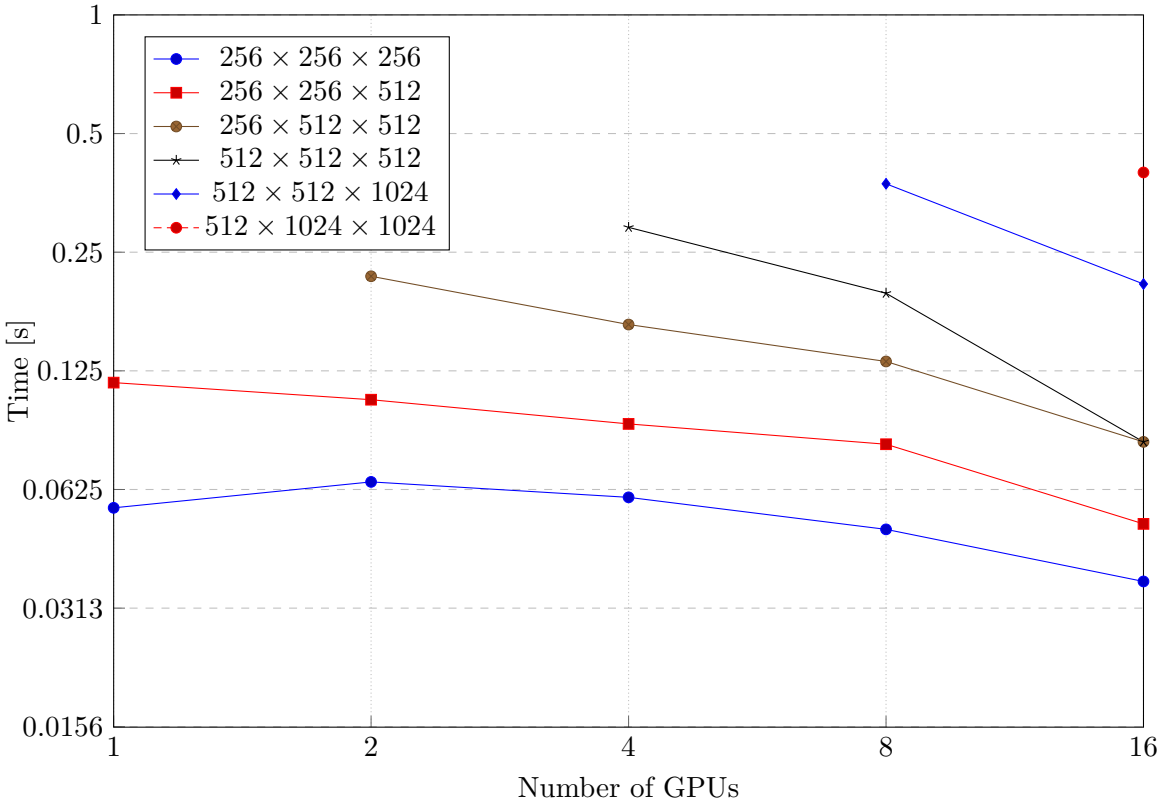
7.3 Computation to communication ratio

For this test domain size 512^3 was chosen as the largest domain size which fits 4 GPUs except for border width 32. For this case domain size was decreased to $256 \times 512 \times 512$. Time of `cudaMemcpy()` was taken from `OVERLAP_COPY [Total]` value from profiler report. Time of MPI communication was taken from `OVERLAP_XCHG_WAIT [Total]`. Time of computation was taken from `SIM_STEP [Mean]` value subtracted by the two previous values. Figure 7.6 and 7.7 shows the result. The left column represents original version and right column CUDA-Aware MPI version.

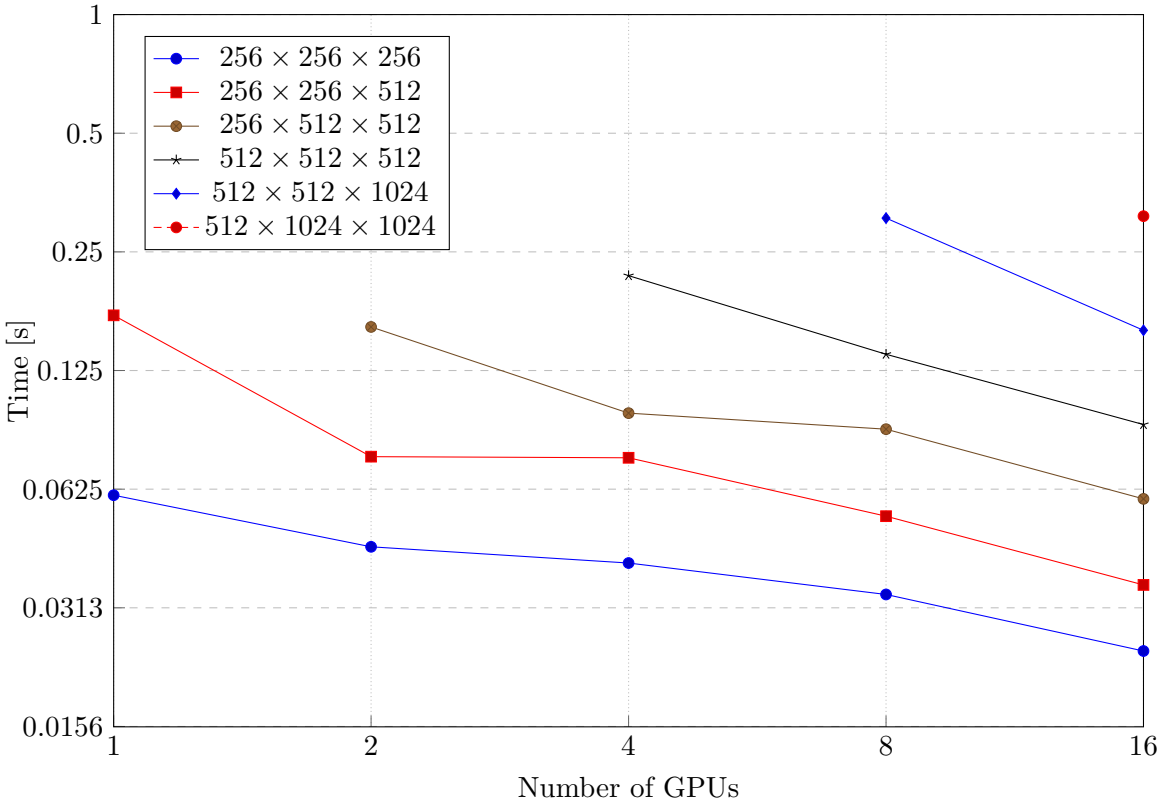
7.4 Uneven load of compute nodes

K-Wave profiler reports timings for every MPI process independently. While processing data from my measurements I noticed that load on each GPU is not spread evenly. This is not specific to CUDA-Aware MPI version. Version with regular MPI behaves in the same way. This behaviour might not be related to grid partitioning, as this happens when the grid is partitioned equally and also unequally.

Graphs 7.9 and 7.10 compare computation and communication time among each GPU within one simulation with domain size 512^3. Simulations were performed with CUDA-Aware MPI support. Average time of one simulation step was taken. Graph 7.9 shows simulation run on 8 GPUs with 2, 2, 2 dimension partitioning. Graph 7.10 shows simulation run on 16 GPUs with 2, 2, 4 dimension partitioning.

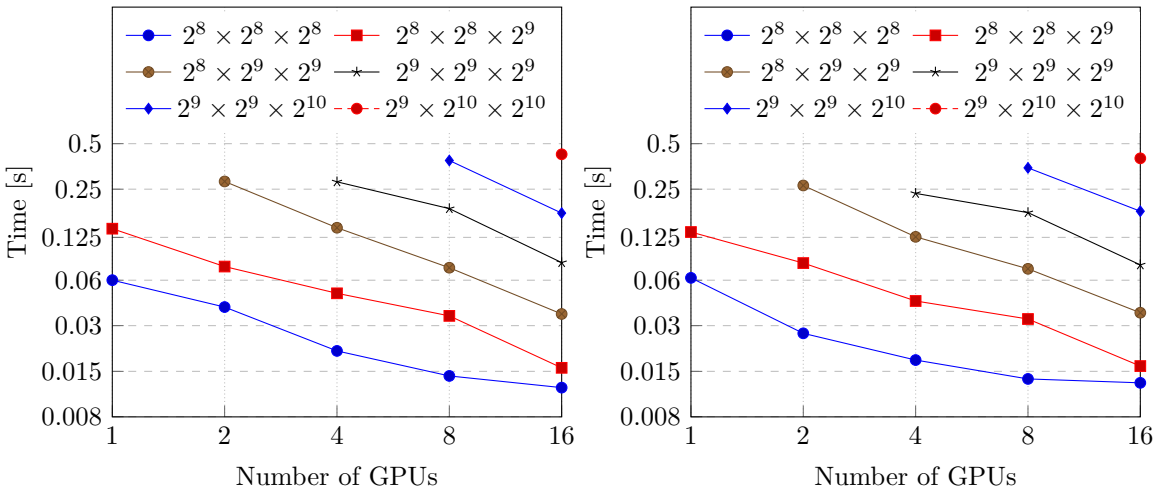


(a) Original version.



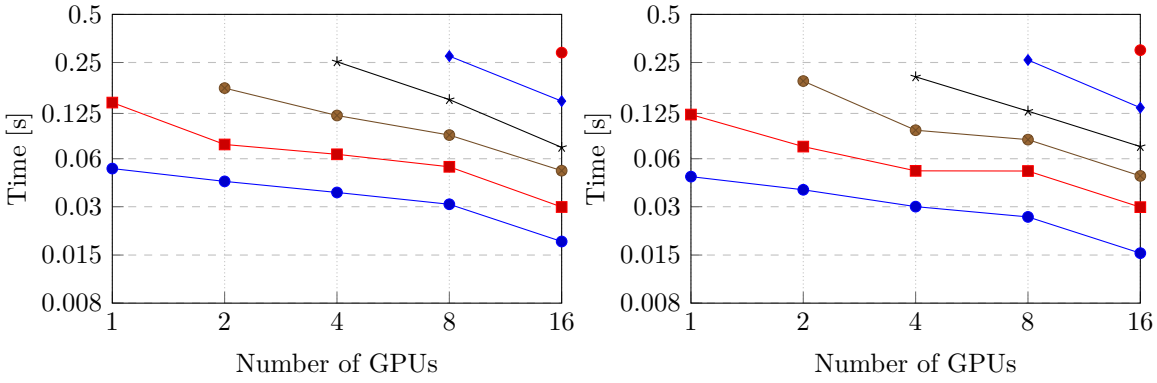
(b) CUDA-Aware MPI version.

Figure 7.1: Strong scaling on Anselm with border size 16.



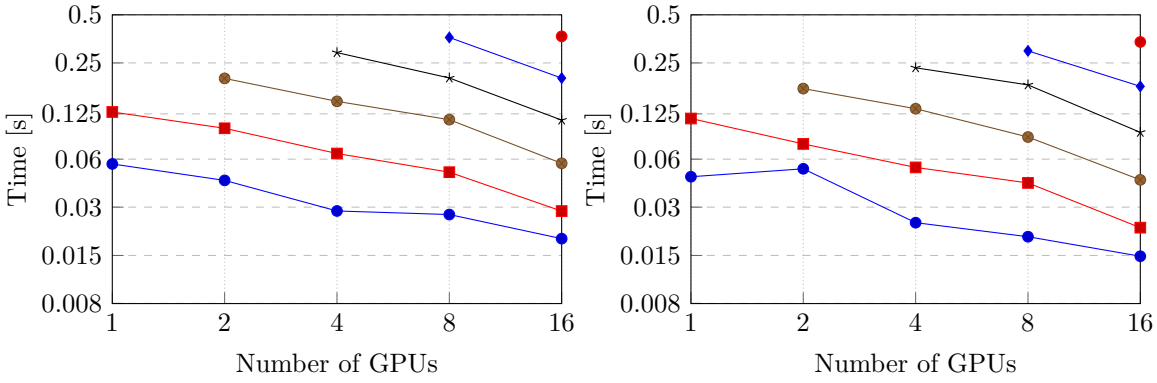
(a) Original version. (b) CUDA-Aware MPI version.

Figure 7.2: Strong scaling on Anselm with border size 2.



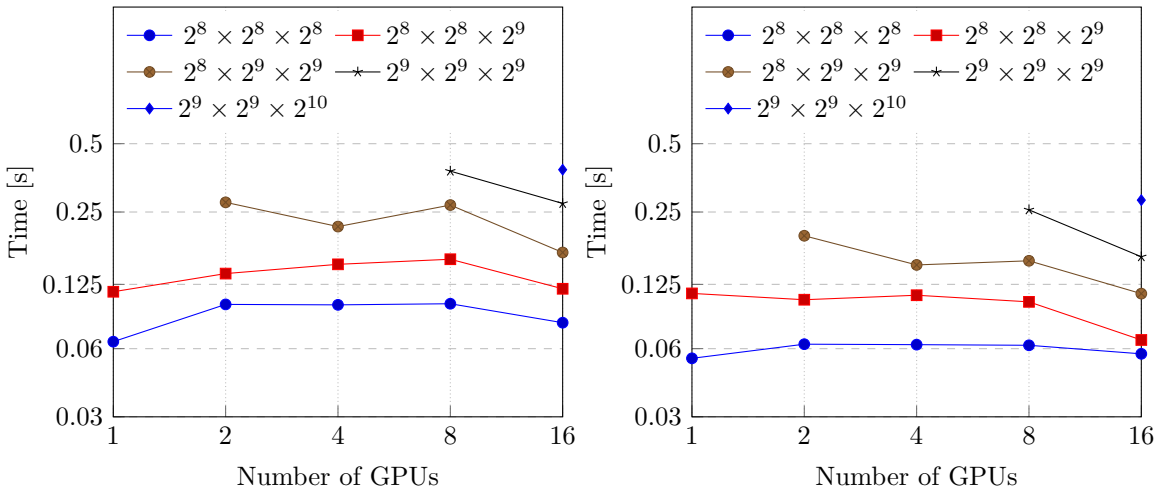
(a) Original version. (b) CUDA-Aware MPI version.

Figure 7.3: Strong scaling on Anselm with border size 4. (Legend is the same as in figure 7.2.)



(a) Original version. (b) CUDA-Aware MPI version.

Figure 7.4: Strong scaling on Anselm with border size 8. (Legend is the same as in figure 7.2.)



(a) Original version

(b) CUDA-Aware MPI version.

Figure 7.5: Strong scaling on Anselm with border size 32.

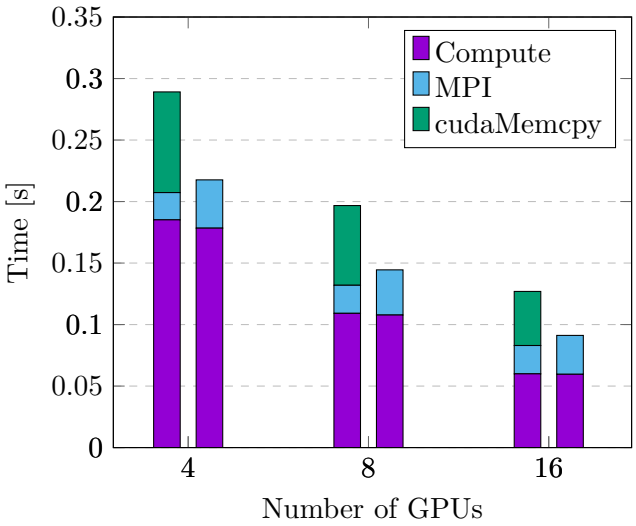
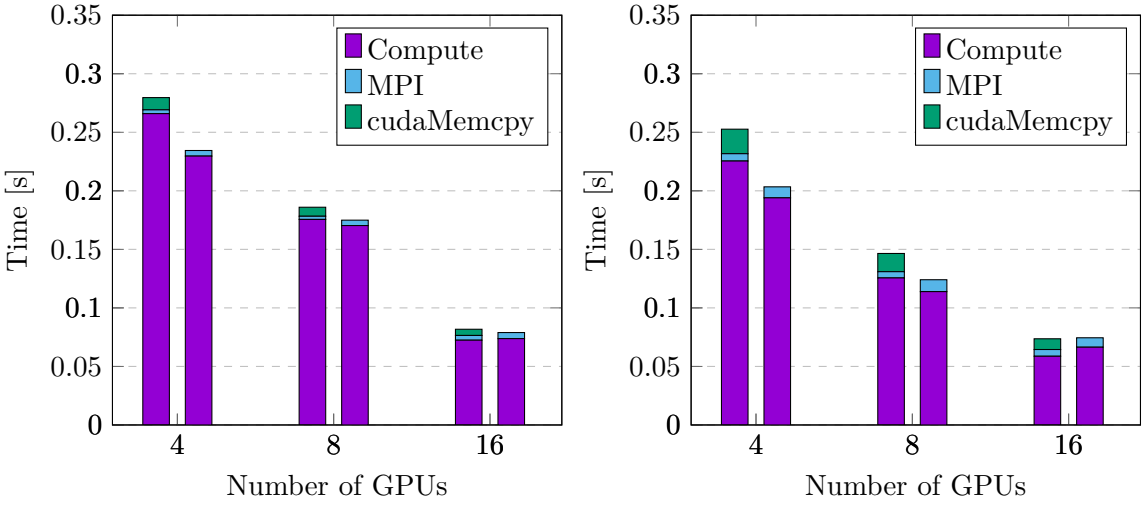
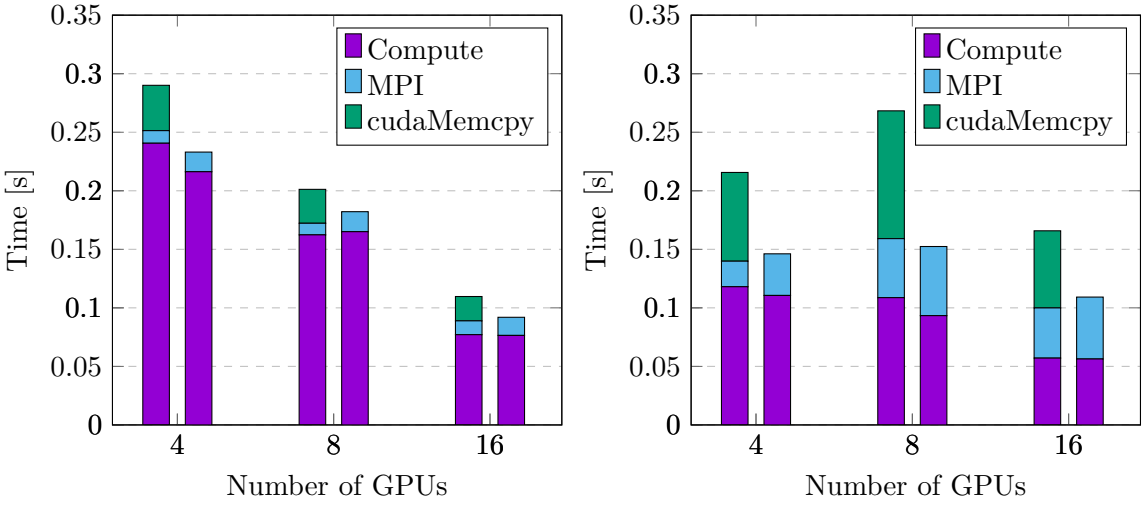


Figure 7.6: Communication to computation ratio for domain size 512^3 and border width 16. Column on the left side represents original version. On the right CUDA-Aware MPI version is displayed.



(a) Border width 2.

(b) Border width 4.



(c) Border width 8.

(d) Border width 32.

Figure 7.7: Communication to computation ratio for domain size 512^3 on Anselm. Except the case with border width 32 where domain size $256 \times 512 \times 512$ is used. Column on the left side represents original version. On the right CUDA-Aware MPI version is displayed.

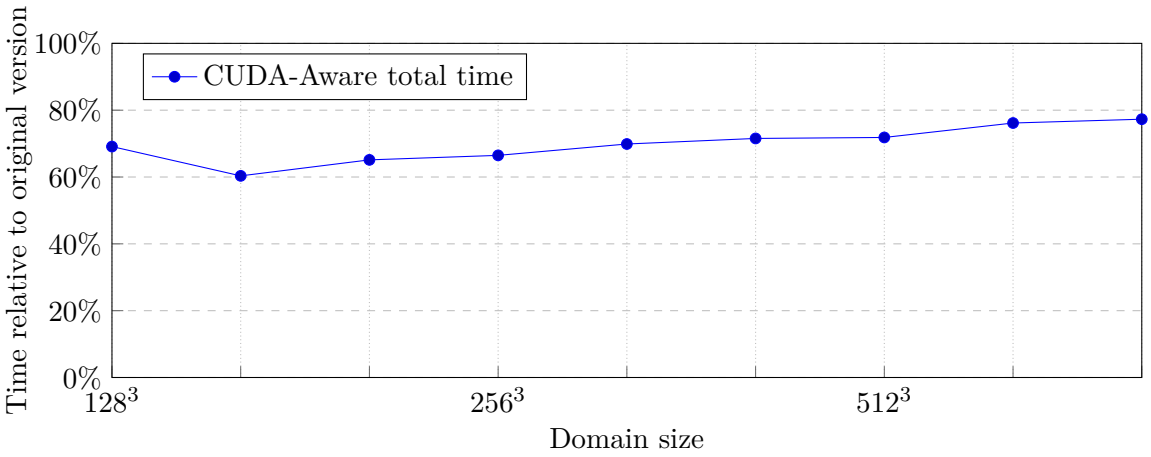


Figure 7.8: Overall computation time of the CUDA-Aware MPI version relative to original version for domain size from 128^3 to $512 \times 1024 \times 1024$ with border width 16 run on 16 GPUs.

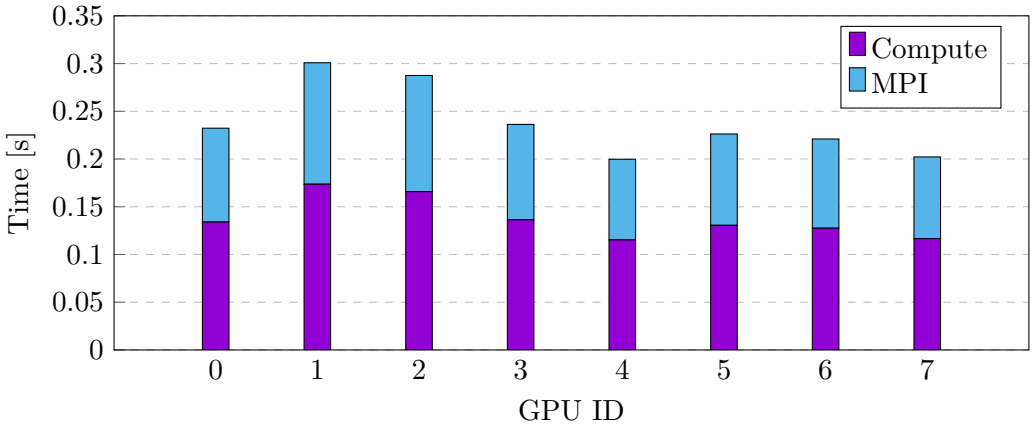


Figure 7.9: Uneven GPU load for domain size $512 \times 512 \times 512$ and border width 16 on 8 GPUs of CUDA-Aware MPI version.

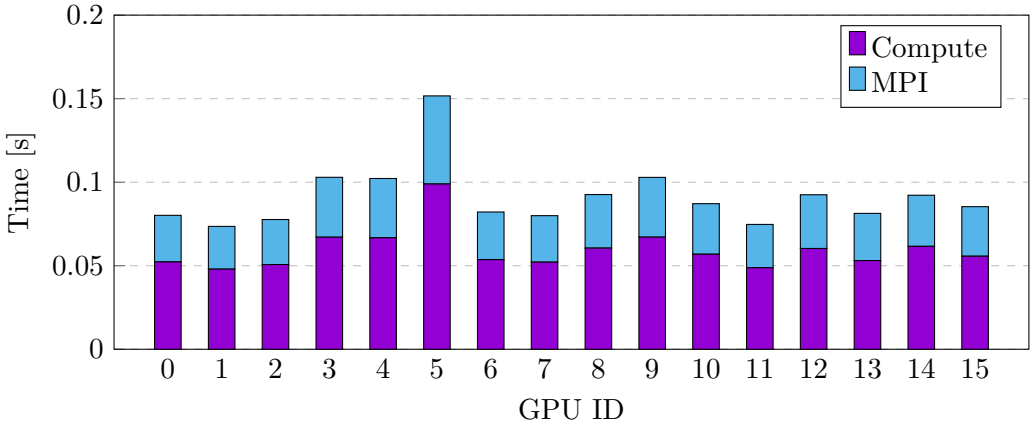


Figure 7.10: Uneven GPU load for domain size $512 \times 512 \times 512$ and border width 16 on 16 GPUs of CUDA-Aware MPI version.

Chapter 8

Single node performance

This chapter shows the single node performance of k-Wave application using both integrated technologies: CUDA-Aware MPI and P2P with CUDA IPC. These performance measurements were conducted on SC-GPU1 and Kinsler.

8.1 SC-GPU1

Measurements on SC-GPU1 were conducted with the same parameters as on Anselm. The difference was that only 2 and 4 GPUs were used. Domain decomposition for 2 GPUs was 1, 1, 2 and for 4 GPUs – 1, 2, 2.

CUDA-Aware MPI parameter `btl_smcuda_cuda_max_send_size` was set to 64 MB and parameter `btl_smcuda_min_size` was set to 4 GB when performing these measurements. CUDA synchronization in k-Wave profiler was turned off during the tests of CUDA-Aware version and CUDA IPC version because it did not allow the IPC version to overlap communication and computation. When performing measurements of the original version, this synchronization was turned on because the profiler was not outputing correct timing data.

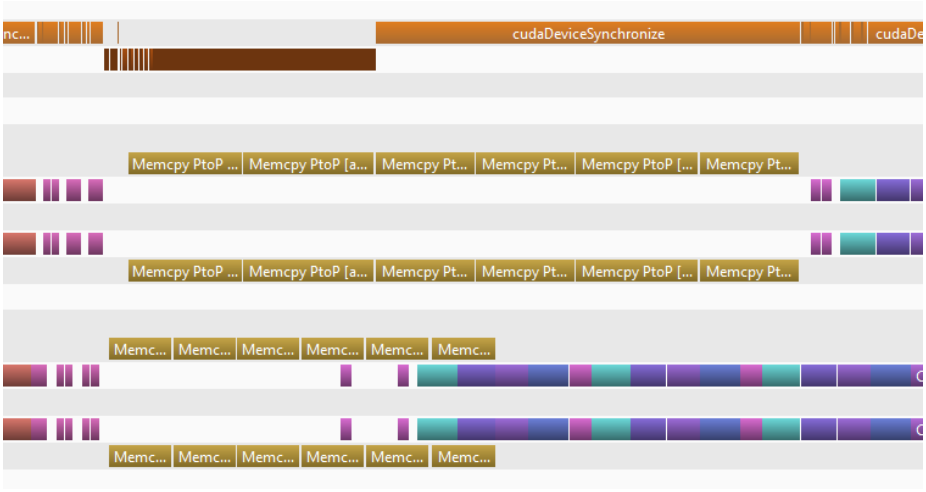


Figure 8.1: Trace from Nvidia Visual Profiler of execution of k-Wave data exchange comparing CUDA-Aware MPI version at the top to the IPC version at the bottom of the trace.

Figure 8.1 shows a trace from Nvidia Visual profiler. In the top part of the figure a data transfer of CUDA-Aware MPI version is shown running synchronously. On the

other hand in the bottom part of the trace CUDA IPC version is shown. It transfers data asynchronously and enables a small communication and computation overlap. The transfer rate of CUDA-Aware MPI version is approximately 5 GBps. The transfer rate of CUDA IPC version is around 10 GBps. Figure 8.2 and 8.3 show communication to computation ratio for 2 GPUs and 4 GPUs.

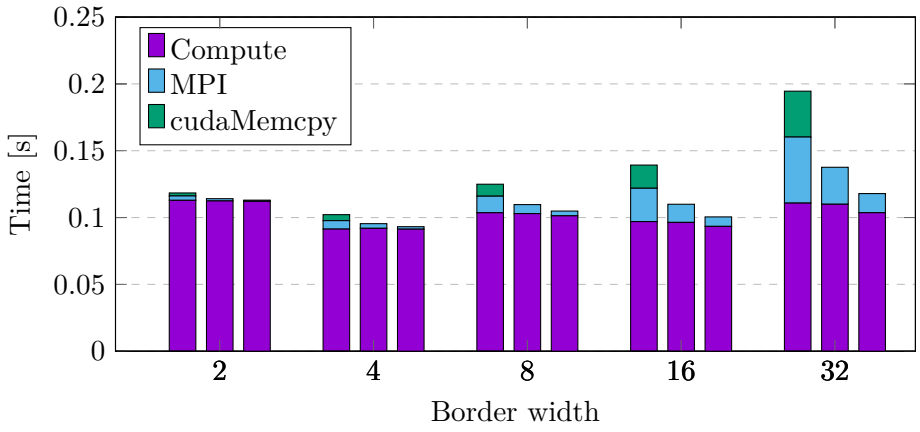


Figure 8.2: Communication to computation ratio for domain size $256 \times 512 \times 512$ and 2 GPUs on SC-GPU1. Column on the left side represents original version. Column in the middle shows CUDA-Aware MPI version. Right column displays CUDA-Aware MPI version with IPC.

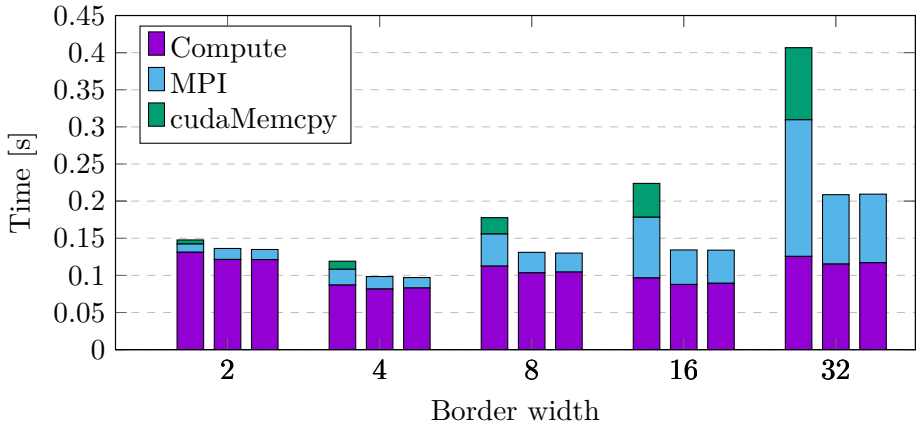


Figure 8.3: Communication to computation ratio for domain size 512^3 and 4 GPUs on SC-GPU1. Column on the left side represents original version. Column in the middle shows CUDA-Aware MPI version. Right column displays CUDA-Aware MPI version with IPC.

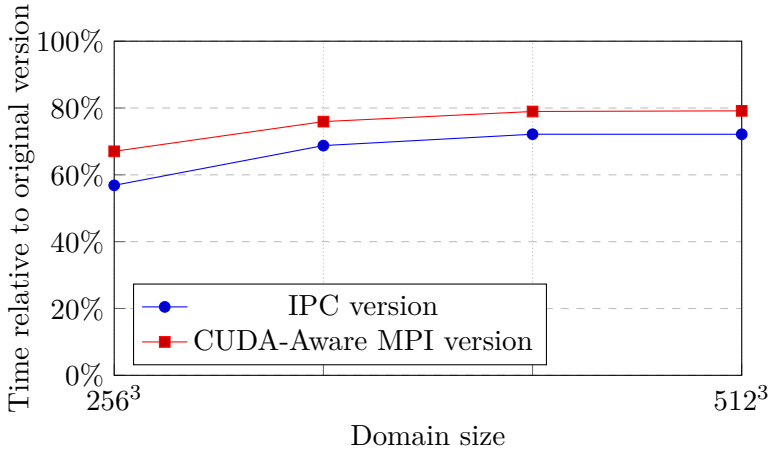


Figure 8.4: Overall computation time of the CUDA-Aware MPI version and IPC version relative to original version for domain sizes from 256^3 to 512^3 with border width 16 run on SC-GPU1 with 2 GPUs.

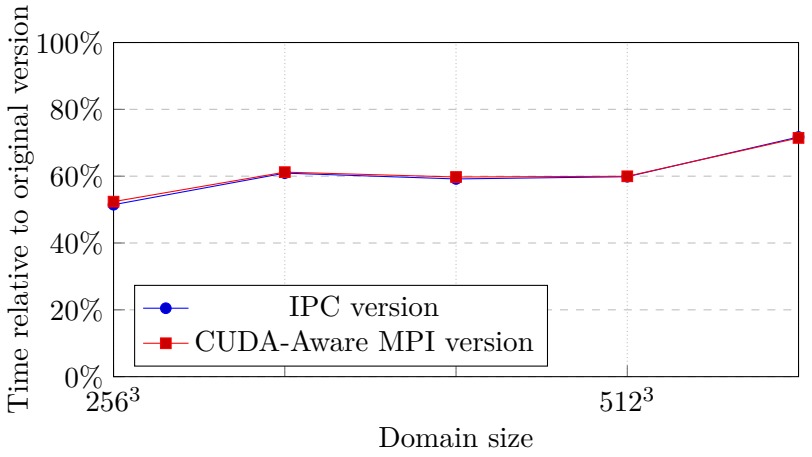


Figure 8.5: Overall computation time of the CUDA-Aware MPI version and IPC version relative to original version for domain sizes from 256^3 to $512 \times 512 \times 1024$ with border width 16 run on SC-GPU with 4 GPUs.

8.2 Kinsler

Measurements on Kinsler were conducted in a similar way to SC-GPU1. Domain decomposition used for the measurements was: 2 GPUs was 1, 1, 2; 4 GPUs – 1, 2, 2 and 8 GPUs – 2, 2, 2. CUDA-Aware MPI parameters were set in the same way as on SC-GPU1: CUDA maximum send size set to 64 MB and shared memory to 4 GB. Figure 8.6 shows communication to computation ratio on 2 GPUs for domain size 512^3 with different border widths. On 2 GPUs the influence of pure P2P transfers is not that significant. The CUDA IPC version is just slightly faster compared to CUDA-Aware MPI version.

Important thing to mention is that a small mistake occurred while performing measurements. In order to achieve the computation and communication overlap of CUDA IPC version the k-Wave profiler synchronization was turned of. Because of this the execution

time of the original version is lower than it should be. This missing computation time is included in cudaMemcpy part. Overall execution time of the original version is correct.

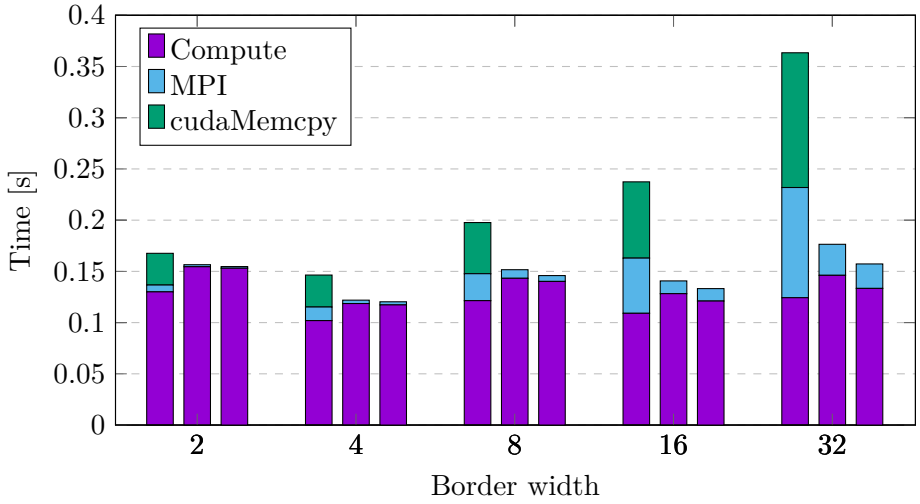


Figure 8.6: Communication to computation ratio for domain size 512^3 for 2 GPUs on Kinsler. Column on the left side represents original version. Column in the middle shows CUDA-Aware MPI version. Right column displays CUDA-Aware MPI version with IPC.

Figure 8.7 shows communication to computation ratio on 4 GPUs for domain size $512 \times 1024 \times 1024$. Compared to previous graph, the domain size increased. This means that also the amount of communication increased. This can be observed on the performance of the original version which one simulation step takes almost one second with border width 32 points. Also the IPC version has the advantage of full P2P access.

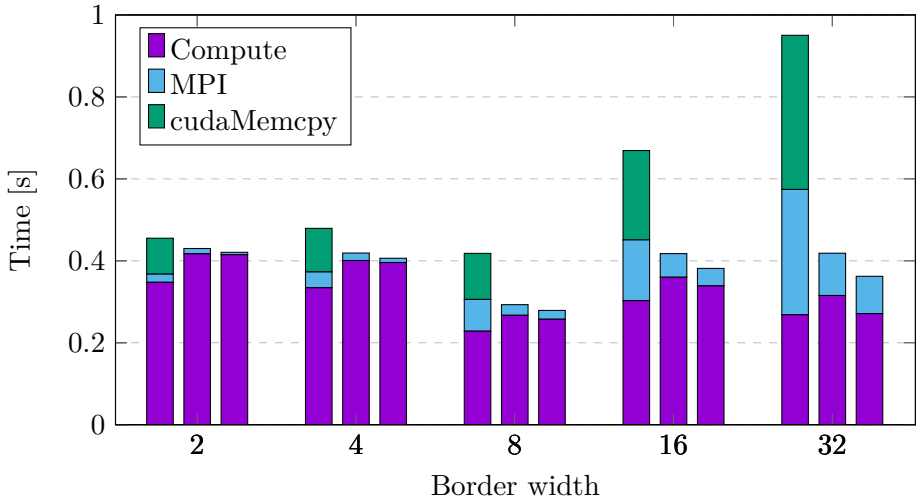


Figure 8.7: Communication to computation ratio for domain size $512 \times 1024 \times 1024$ for 4 GPUs on Kinsler. Column on the left side represents original version. Column in the middle shows CUDA-Aware MPI version. Right column displays CUDA-Aware MPI version with IPC.

Figure 8.8 shows communication to computation ratio on 8 GPUs for domain size $512 \times 1024 \times 1024$. Because the domain is partitioned into more GPUs the computation time decreased compared to previous case. But the amount of communication increased significantly. It is also noticeable that CUDA IPC version takes almost the same time as CUDA-Aware MPI version.

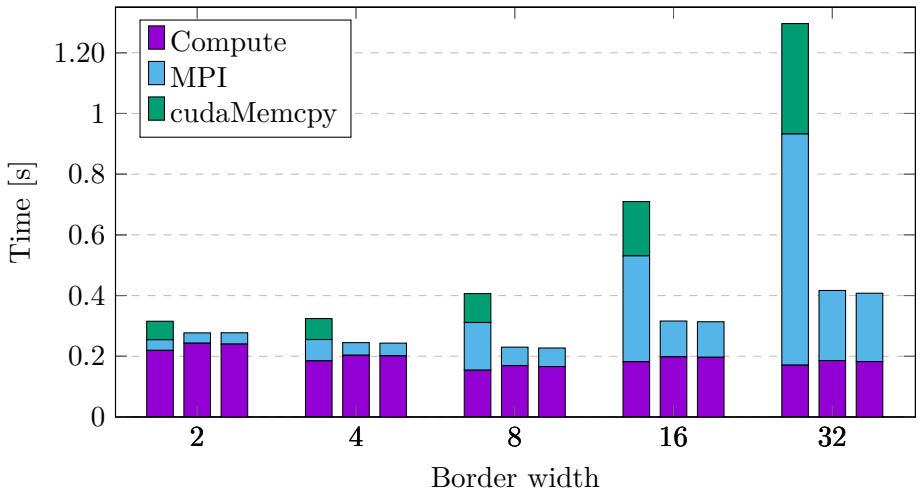


Figure 8.8: Communication to computation ratio for domain size $512 \times 1024 \times 1024$ for 8 GPUs on Kinsler. Column on the left side represents original version. Column in the middle shows CUDA-Aware MPI version. Right column displays CUDA-Aware MPI version with IPC.

Figure 8.9 shows the time of one simulation step of both optimized versions relative to the time of original version for 4 GPUs. The CUDA IPC can take advantage of the full P2P connection and it performs approximately 5 % faster compared to CUDA-Aware MPI version. In figure 8.10 the IPC version is only 1 % faster than CUDA-Aware MPI version running on 8 GPUs.

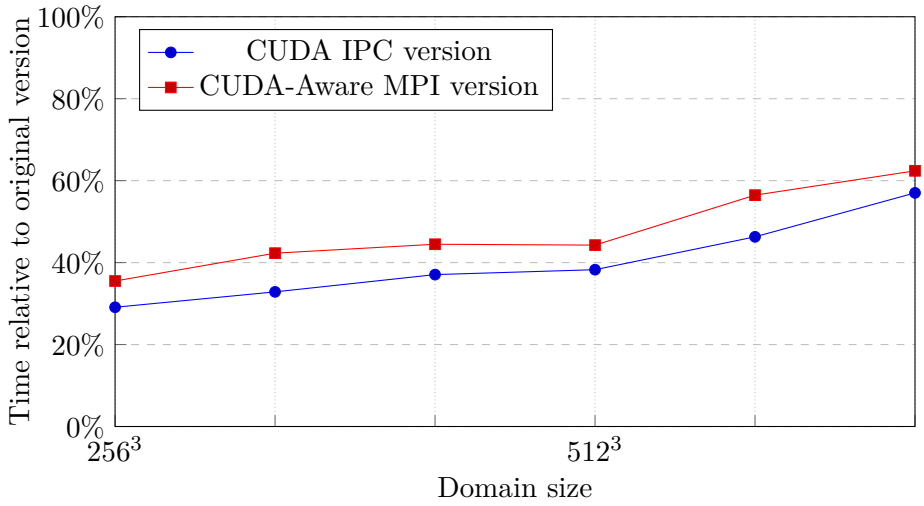


Figure 8.9: Overall computation time of the CUDA-Aware MPI version and IPC version relative to original version for domain sizes from 256^3 to $512 \times 1024 \times 1024$ with border width 16 run on Kinsler with 4 GPUs.

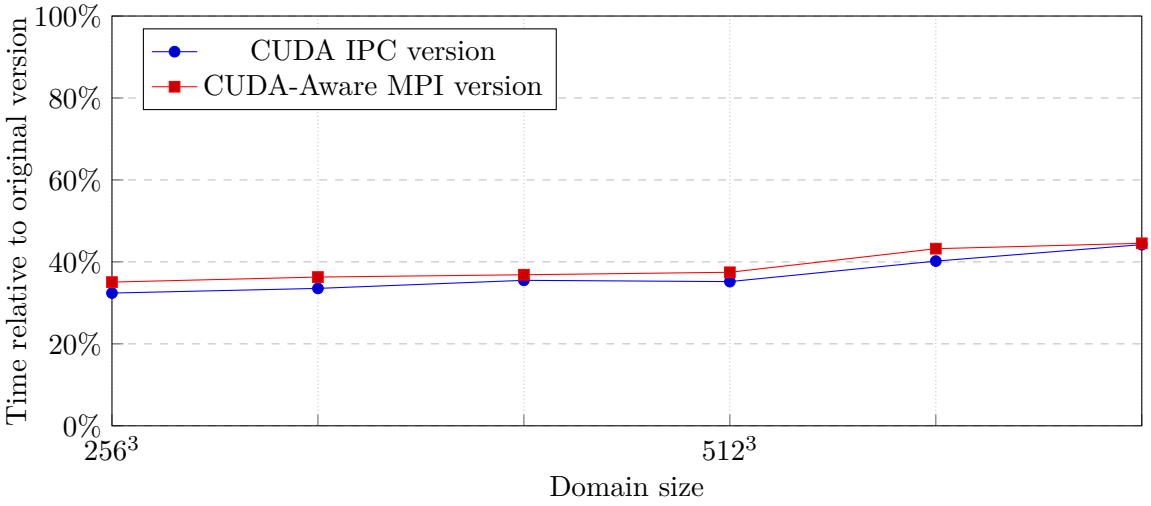


Figure 8.10: Overall computation time of the CUDA-Aware MPI version and IPC version relative to original version for domain sizes from 256^3 to $512 \times 1024 \times 1024$ with border width 16 run on Kinsler with 8 GPUs.

Chapter 9

Future development

This chapter discusses suggestions for the future development of this thesis. These techniques can be used in any application but the main focus is on k-Wave. The first section explains direct data exchange between GPUs using CUDA UVA. This approach does not require data serialization into separate buffers. The second section describes the use of interconnect with higher bandwidth than PCI-E 3.0, specifically NVlink and PCI-E 4.0.

9.1 Direct data copy

So far, my thesis did not mention the use of CUDA Unified Virtual Addressing technology (UVA). This might be the key technology for further performance improvement of k-Wave and also applications using similar computation and communication pattern. K-Wave needs to exchange border and corner data between local computational matrices stored as 3D array. The current solution to this is to extract data from the local matrix to a buffer. Then, the data are sent to receiving GPU. After that the data are injected from the buffer into the local matrix in the receiving GPU. The advantage of this approach is that the data are copied to receiving GPU in one bulk transaction.

CUDA UVA allows the exchanged data to be copied between local matrices in different GPUs directly without the use of separate buffers. This is because a memory pointer from receiving GPU can be used in kernel calls of sending GPU as if it was local memory pointer. The first option would be to utilize the `cudaMemcpy3D()` function. This function and its derivatives are suitable to perform data exchanges between local matrices allocated in different GPUs. The second possibility is to modify the extraction kernels. Instead of serializing data into the send buffer these modified kernels would copy data from local computation matrix directly into computation matrix in receiving GPU. In both cases, it is necessary to apply the Bell function to transferred data. According to the k-Wave's trace from Nvidia Visual Profiler, it takes up to 10 % of computation time to perform extraction and injection kernels. This time could be saved by leaving these kernels out. The percentage depends on domain size, domain decomposition and border width.

Current k-Wave version optimized with CUDA IPC would allow these direct transfers only within GPUs connected to a single CPU. This is because IPC can export memory pointer only if P2P access is available. In order to enable direct transfers over QPI interconnect it is necessary to use shared address space. This can be achieved by exchanging MPI processes with OpenMP threads. It would mean that each thread would control single GPU. Threads have a common address space and memory pointers can be shared within

each other. This modification would enable direct transfers to all GPUs in a single node environment.

9.2 Use of NVlink

Another possibility to increase GPU-to-GPU data transfer speed is to utilize NVlink. It is an interconnection technology developed by NVIDIA and it can provide bandwidth several times higher than PCI-E 3.0. This interconnect can be utilized by P2P transfers between GPUs in single node environment. Unfortunately, NVlink is only supported by the highest professional class GPU models from NVIDIA. For this reason, servers supporting this technology are harder to access and there was no way for me to test its influence.

In the following few years, it is expected from GPU and motherboard manufacturers to use PCI-E 4.0 interface instead of version 3.0. PCI-E 4.0 doubles the bandwidth compared to PCI-E 3.0. PCI-E 4.0 will be probably introduced first in professional class GPUs and later in consumer class GPUs but eventually it will mean overall data transfer speedup.

Chapter 10

Conclusion

In my master thesis, I was researching methods on effective communication between GPUs. Examined methods are for inter-node and also intra-node communication. Effective means that data transfers involve less CPU, system buffers can be shared with two different devices and there is less data staging in the host memory.

On Anselm I find beneficial to use CUDA-Aware MPI when conducting inter-node GPU-to-GPU transfers. There was a significant performance improvement compared to regular MPI performance. Unfortunately, I was not able to test the influence of RDMA as it is not supported on Anselm. On SC-GPU1 transfers with peer-to-peer access represent an improvement compared to regular data transfer within single PCI-E root complex. When performing inter-socket GPU-to-GPU data transfer, the QPI link represents significant overhead. The performance of CUDA-Aware MPI is not as high as the performance of CUDA transfer routines.

Furthermore, k-Wave toolbox was introduced. Its multi-GPU implementation which uses local domain decomposition was accelerated using CUDA-Aware MPI. P2P was also integrated into k-Wave using CUDA Inter-process communication. In a distributed environment of Anselm supercomputer using 16 grid points border width, the accelerated solution can perform 20 – 30% faster compared to the original version.

On a single node machine SC-GPU1 depending on the simulation parameters the overall simulation time of CUDA-Aware MPI version is 10 – 30% shorter. CUDA IPC version can only benefit from full P2P access. When run on 2 GPUs it performs even 5 % faster than CUDA-Aware MPI version. On Kinsler server with 8 GPUs the execution time of both optimized versions can be up to 60 % shorter running with 16 points wide borders.

Bibliography

[1] *Výpočetní cluster FIT VUT*. FIT VUT v Brně. [Online; visited 06.01.2018]. Retrieved from: http://www.fit.vutbr.cz/CVT/cluster/

[2] Cheng, J.; Grossman, M.; McKercher, T.: *Professional CUDA C Programming*. John Wiley & Sons, Inc.. 2014. ISBN 978-1-118-73932-7.

[3] Corp., N.: *GeForce GTX 1080 Specifications*. [Online; visited 09.01.2018]. Retrieved from: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1080/specifications

[4] Ghent University: *EasyBuild Documentation Release 20171017.01*. December 2017. [Online; visited 05.01.2018]. Retrieved from: https://media.readthedocs.org/pdf/easybuild/latest/easybuild.pdf

[5] GIGA-BYTE Technology Co.: *Gigabyte G250-G51 User Manual*. October 2016. [Online; visited 24.07.2018]. Retrieved from: http://download.gigabyte.eu/FileList/Manual/server_manual_g250-s88_e_11.pdf

[6] HPC Advisory Council: *Introduction to High-Speed InfiniBand Interconnect*. [Online; visited 07.01.2018]. Retrieved from: http://www.hpcadvisorycouncil.com/pdf/Intro_to_InfiniBand.pdf

[7] Intel Corp.: *Intel Xeon Processor E5–2600 v4 Product Family Specification Update*. July 2017. [Online; visited 24.07.2018]. Retrieved from: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v4-spec-update.pdf

[8] Intel Corp.: *Intel Xeon Processor E5 v3 Product Family Specification Update*. September 2017. [Online; visited 08.01.2018]. Retrieved from: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf

[9] IT4Innovations: *Co je IT4Innovations*. [Online; visited 07.01.2018]. Retrieved from: http://www.it4i.cz/o-nas/co-je-it4innovations/

[10] Jaroš, J.; Rendell, P. A.; Treeby, E. B.: Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications*. vol. 2016, no. 2.

2016: pp. 137–155. ISSN 1741-2846.
Retrieved from:
http://hpc.sagepub.com/content/early/2015/04/28/1094342015581024.full.pdf

[11] Jaroš, J.; Vaverka, F.; Treeby, E. B.: Spectral Domain Decomposition Using Local Fourier Basis: Application to Ultrasound Simulation on a Cluster of GPUs. *International Journal of Supercomputing Frontiers and Innovations*. vol. 3, no. 3. 2016: pp. 39–54. ISSN 2313-8734.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=11149

[12] Kraus, J.: *An Introduction to CUDA-Aware MPI*. March 2013. [Online; visited 10.01.2018].
Retrieved from:
https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/

[13] Kraus, J.; Messmer, P.: *Multi GPU Programming With MPI*. 2014. [Online; visited 09.01.2018].
Retrieved from: http://on-demand.gputechconf.com/gtc/2014/presentations/S4236-multi-gpu-programming-mpi.pdf

[14] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard Version 3.1*. June 2015. [Online; visited 09.01.2018].
Retrieved from: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[15] Morgan, T. P.: *HPC Poised For Big Changes, Top To Bottom*. June 2017. [Online; visited 06.01.2018].
Retrieved from: https://www.nextplatform.com/2017/06/19/hpc-poised-big-changes-top-bottom/

[16] NVIDIA Corp.: *Tesla K20 GPU Accelerator*. November 2012. [Online; visited 08.01.2018].
Retrieved from: https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf

[17] NVIDIA Corp.: *Tesla P40 GPU Accelerator*. November 2016. [Online; visited 24.07.2018].
Retrieved from:
http://images.nvidia.com/content/tesla/pdf/Tesla-P40-Product-Brief.pdf

[18] PCI-SIG: *PCI Express – 3.0 Frequently Asked Questions*. [Online; visited 08.01.2018].
Retrieved from: https://pcisig.com/faq

[19] Rupp, K.: *CPU, GPU and MIC Hardware Characteristics over Time*. August 2016. [Online; visited 05.01.2018].
Retrieved from: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[20] Sliva, R.; Stanek, F.: *Best Practice Guide Anselm*. IT4Innovations, VSB – Technical University of Ostrava. May 2013. [Online; visited 06.01.2018].
Retrieved from:
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Anselm.pdf

[21] Super Micro Computer, Inc.: *Supermicro X10DRG-Q Motherboard User's Manual Revision 1.2a*. September 2017. [Online; visited 05.01.2018].
Retrieved from:
http://supermicro.com/manuals/motherboard/C612/MNL-1677.pdf

[22] TOP500.org: *Top500 List - June 2017*. June 2017. [Online; visited 07.01.2018].
Retrieved from: https://www.top500.org/lists/2017/06/

[23] Vaverka, F.: Case Study on Multi-domain Decomposition of k-Wave Simulation Framework. In *Computer achitectures and diagnostics 2016*. Faculty of Information Technology BUT. 2016. ISBN 978-80-214-5376-0. pp. 37–40.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=11200